NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A PRO-ACTIVE ROUTING PROTOCOL FOR CONFIGURATION OF SIGNALING CHANNELS IN SERVER AND AGENT-BASED ACTIVE NETWORK MANAGEMENT (SAAM)

by

Hasan Akkoc

June 2000

Thesis Advisor:	Geoffrey Xie
Second Reader:	Deborah Kern

Approved for public release; distribution is unlimited.

QUALITY INSPECTED 4

20000721 047

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE: A PRO–ACTIVE ROUTING PROTOCOL FOR CONFIGURATION OF SIGNALING CHANNELS IN SERVER AND AGENT BASED ACTIVE NETWORK MANAGEMENT (SAAM).	5. FUNDING NUMBERS
6. AUTHOR(S) Akkoc, Hasan	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA and NASA	10. SPONSORING / MONITORING AGENCY REPORT NUMBER G417

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited.	Statement A

13. ABSTRACT *(maximum 200 words)*

As networks are upgraded to provide services for streaming applications, the current way of routing is not satisfactory. Server and Agent based Active network Management (SAAM) introduces a novel network architecture that provides guaranteed quality of services to real-time traffic. In SAAM, the server and routers need to establish two-way, robust, and efficient signaling channels for exchange of control and management information. Any change in network topology must be determined and handled as they occur in order to support guaranteed services. Local detection of topological changes and hop-by-hop dissemination of knowledge of these changes is not optimal for SAAM architecture. A reactive method of updating routing tables takes longer time than tolerable for real-time traffic. Therefore, a pro-active approach that reconfigures the signaling channels in real time and without degrading services to user traffic is mandatory. This thesis presents such a pro-active routing protocol for configuring the signaling channels of a SAAM region.

14. SUBJECT TERMS Routing Protocol, Signaling Channel Configuration, Soft-State Approach, Pro-Active Approach, Networks	15. NUMBER OF PAGES 243
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18298-102

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A PRO-ACTIVE PROTOCOL FOR CONFIGURATION OF SIGNALING CHANNELS IN SERVER AND AGENT BASED ACTIVE NETWORK MANAGEMENT (SAAM)

Hasan Akkoc
First Lieutenant, Turkish Army
B.S., Middle East Technical University, 1994

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2000

Author: _____
Hasan Akkoc

Approved by: _____
Geoffrey Xie, Thesis Advisor

_____
Deborah Kern, Second Reader

_____
Dan C. Boger, Chairman
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

As networks are upgraded to provide services for streaming applications, the current way of routing is not satisfactory. Server and Agent based Active network Management (SAAM) introduces a novel network architecture that provides guaranteed quality of services to real-time traffic. In SAAM, the server and routers need to establish two-way, robust, and efficient signaling channels for exchange of control and management information. Any change in network topology must be determined and handled as they occur in order to support guaranteed services. Local detection of topological changes and hop-by-hop dissemination of knowledge of these changes is not optimal for SAAM architecture. A reactive method of updating routing tables takes longer time than tolerable for real-time traffic. Therefore, a pro-active approach that reconfigures the signaling channels in real time and without degrading services to user traffic is mandatory. This thesis presents such a pro-active routing protocol for configuring the signaling channels of a SAAM region.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1

A BACKGROUND 1

B. OVERVIEW OF SAAM 2
1. The Logical Model Of SAAM 3
2. Hierarchical Organization Of SAAM Architecture 4

C. SCOPE OF THIS THESIS 6

D. MAJOR CONTRIBUTIONS OF THIS THESIS 6

E. ORGANIZATION OF THIS THESIS 6

II. RELATED TOPICS 9

A. ROUTING PROTOCOLS 9
1. Robustness 10
2. Manageable Signaling Overhead 11
3. Path Optimization 11
4. Manageable Routing Table Size 11

B. DISTINCT PROPERTIES OF SAAM ARCHITECTURE RELATED TO SIGNALING CHANNELS 12
1. Asymetrical Flow Of Signaling Traffic 12
2. Pro-Active Response To Topological Changes 12

C. INTERIOR ROUTING PROTOCOLS 13
1. Distance-Vector Algorithm 13
2. Link-State Algorithm 15
3. Suitability Of Interior Routing Protocols For SAAM 17

D. MULTICAST ROUTING PROTOCOLS 17
1. Group-Shared Tree Approach 18
2. Source-Based Tree Approach 19
3. Why Multicast Routing Protocols Can't Be Used For SAAM 21

III. SAAM SIGNALING CHANNEL CONFIGURATION PROTOCOL DESIGN 23

A. INTRODUCTION TO SIGNALING CHANNEL CONFIGURATION PROTOCOL 23
1. The Spanning Tree Approach 25
2. How To Make A Two -Way Spanning Tree 26

B. HOW SCCP HANDLES TOPOLOGY CHANGES AS THEY OCCUR 30

C. MAJOR ISSUES THAT HAVE INFLUENCED DESIGN OF SCCP 31
1. Issues Related to Server-Bound Signaling Channels 31
2. Issues Related to Router-Bound Signaling Channel 31
3. Routing Tables Used By SCCP 31

D. SCCP ALGORITHM 33

E. APPLICATION OF SCCP TO A SAMPLE TOPOLOGY 35

IV. SAAM SIGNALLING CHANNEL CONFIGURATION PROTOCOL IMPLEMENTATION 41

A. SAAM SIGNALING MODEL BEFORE IMPLEMANTATION OF SCCP 41

B. MESSAGES ADDED TO EXISTING SAAM EMULATION SOFTWARE FOR DEPLOYMENT OF SCCCP 41
1. Packet Formats Used In SAAM Environment 42
2. Integration of Messages Introduced By SCCP to SAAM Packet Format 45
3. Messages Designed To Add Flexibility To SCCP 49

C. CLASSES ADDED TO SAAM EMULATION SOFTWARE FOR IMPLEMENTION OF SCCP 52
1. ServerInformation Class 52
2. AutoConfigurationExecutive Class 54
3. TimerHandler Class 57
4. ServerTableEntry Class 57
5. ServerTable Class 58
6. RouterBoundCtrlChTableEntry Class 57
7. RouterBoundCtrlChTable Class 58

D. MODIFIED CLASSES 58
1. ControlExecutive Class 59
2. PacketFactory Class 61
3. RoutingAlgorithm Class 62
4. ServerAgentSymetric Class 63
5. Server Class 63

V. TEST OF SCCP 65

A. DESCRIPTION OF TEST TOPOLOGY 65

B. TESTING OF SCCP 67
1. Initialization of Test-Bed 68
2. Execution of SCCP for A Single Configuration Cycle 72
3. Configured Signaling Channels 80
4. Results of Periodical Execution 81

VI. CONCLUSIONS 83

A. SYNOPSIS AND CONCLUSION 83

B. LESSONS LEARNED 84
1. Importance Of Coordination 84
2. Router Id Implementation 84

C. FUTURE WORK 85
1. Determination of Refresh Interval 85
2. Determination of Local and Global Timers 85
3. Implementation Of SCCP With Different Metrics For Configuration of Signaling Channels 85
4. Determination of Inter-Region Signaling Channels 86

APPENDIX A. DEFINITIONS OF NODES IN ROOTED TREES 87

APPENDIX B. SERVER STATE DIAGRAM 89

APPENDIX C. ROUTER STATE DIAGRAM 91

APPENDIX D. CLASSES ADDED TO SAAM FOR SCCP DEPLOYMENT 93

APPENDIX E. CLASSES MODIFIED IN SAAM FOR SCCP DEPLOYMENT 138

LIST OF REFERENCES 223

INITIAL DITRIBUTION LIST 225

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES LIST OF FIGURES

Figure 1.1 Sample SAAM Network Architecture 3
Figure 1.2 Logical Model Of SAAM 4
Figure 1.3 Hierarchical Organization Of SAAM Servers 5
Figure 2.1 A Sample Group-Shared Tree 18
Figure 2.2 A Sample Source-Based Tree 20
Figure 3.1 SAAM Topology Displaying Router-bound Signaling Channels 23
Figure 3.2 SAAM Topology Displaying Server-bound Signaling Channels 24
Figure 3.3 SAAM Topology With Configured Signaling Channels (Thick lines) 28
Figure 3.4 Flow Routing Table Format 32
Figure 3.5 The Format Of RBCCT for Server with Flow Ids 'k' and 'k-1' 33
Figure 3.6 SAAM Topology For SCCP Application 35
Figure 3.7 FlowRoutingTable of Router After Server-Bound Signaling Channel Is Configured 37
Figure 3.8. FlowRoutingTable of Router B After Server-Bound Signaling Channel Is Configured 37
Figure 3.9 FlowRoutingTable of Router C After Server-Bound Signaling Channel Is Configured 37
Figure 3.10 RBCCT Of The Server 38
Figure 3.11 RBCCT Of Router A 39
Figure 3.12 RBCCT Of Router B 39
Figure 3.13 RBCCT Of Router C 39
Figure 3.14 Configured Signaling Channels 39
Figure 4.1 Emulation Packet Format 43
Figure 4.2 Demo Packet Format 43
Figure 4.3 SAAM Packet Format 44
Figure 4.4 DCM Embedded In SAAM Packet 46
Figure 4.5 UCM Embedded In SAAM Packet 47
Figure 4.6 PN Message Embedded In SAAM Packet 48
Figure 4.7 Configuration Message Embedded In SAAM Packet 50
Figure 4.8 TimeScale Message Embedded In SAAM Packet 52
Figure 4.9 Sample ServerTable 57
Figure 4.10 Sample RouterBoundCtrlChTable 58
Figure 5.1 Sample Test Topology 66
Figure 5.2 Emulation Table Of Primary Server 69
Figure 5.3 Emulation Table Of Backup Server 69
Figure 5.4 Emulation Table Of Router A 69
Figure 5.5 Emulation Table Of Router B 69
Figure 5.6 Emulation Table Of Router C 70
Figure 5.7 Emulation Table Of Router D 70
Figure 5.8 ARPCache Table Of Primary Server 70
Figure 5.9 ARPCache Table Of Backup Server 71
Figure 5.10 ARPCache Table Of Router A 71
Figure 5.11 ARPCache Table Of Router B 71
Figure 5.12 ARPCache Table Of Router C 72

Figure 5.13 ARPCache Table Of Router D 72
Figure 5.14 ServerTable Of Primary Server 73
Figure 5.15 ServerTable Of Backup Server 74
Figure 5.16 ServerTable Of Router A 74
Figure 5.17 ServerTable Of Router B 74
Figure 5.18 ServerTable Of Router C 75
Figure 5.19 ServerTable Of Router D 75
Figure 5.20 Server-Bound Signaling Channel Table Of Primary Server 75
Figure 5.21 Server-Bound Signaling Channel Table Of Backup Server 76
Figure 5.22 Server-Bound Signaling Channel Table Of Router A 76
Figure 5.23 Server-Bound Signaling Channel Table Of Router B 76
Figure 5.24 Server-Bound Signaling Channel Table Of Router C 76
Figure 5.25 Server-Bound Signaling Channel Table Of Router D 77
Figure 5.26 Router-Bound Signaling Channel Tables Of Primary Server 77
Figure 5.27 Router-Bound Signaling Channel Tables Of Backup Server 78
Figure 5.28 Router-Bound Signaling Channel Tables Of Router A 78
Figure 5.29 Router-Bound Signaling Channel Tables Of Router B 79
Figure 5.30 Router-Bound Signaling Channel Tables Of Router C 79
Figure 5.31 Router-Bound Signaling Channel Tables Of Router D 80
Figure 5.32 Configured Signaling Channels At The End Of First Cycle 81
Figure 5.33 Router-Bound Signaling Channel Table Of Primary Server After Configuration Cycle Two 82
Figure 5.34 Router-Bound Signaling Channel Table Of Primary Server After Configuration Cycle Two 82

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 5.1 Node Information Of Sample SAAM Topology.....65
Table 5.2 Length Of Messages Employed By SCCP.....72

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENTS/DEDICATIONS

I would like to express my sincere gratitude to my thesis advisor, Professor Geoffrey Xie. Without his constant support I would never have any chance of completing this thesis. I will always remember the light he brought up to me by sincere support and motivation. His style will always be my guide throughout my life. Additionally, I would like to thank CDR Debra Kern for her contributions and support. I would also like to thank to my parents, Emine and Mahmut Akkoc, without whose love and sacrifice I would not have the possibility to come so far in my education. And finally, I would like to dedicate this thesis to two very special women in my life; my fiancé Alessia Deligios and my mother Emine Akkoc for their unconditional love.

I. INTRODUCTION

A. BACKGROUND

The number of nodes connected to the Internet has grown at an explosive rate. The volume and content of network traffic have also changed considerably. While exclusively data twenty-some years ago, the traffic today is composed of image, audio and video beside data.

The traditional Internet provides "best-effort" service to packets without any guarantee of delivery. It can't provide satisfactory support for networked real time applications such as interactive voice and video, which are very sensitive to end-to-end network delay and the delay jitter. These applications usually generate a huge number of bits per second, and the traffic has to be streamed, or transmitted in a smooth continuous flow rather than in bursts [1]. Current router-centric network architectures would be over-extended to meet the Quality of Service (QoS) requirements of real-time applications. So the main challenge of the Next Generation Internet (NGI) is to develop feasible networking solutions that are capable of serving diverse user traffic (i.e., capable of integrated service).

Server and Agent based Active network Management (SAAM) is one of the ongoing research projects under the NGI initiative. The SAAM project is sponsored by National Aeronautics and Space Administration (NASA) and Defense Advanced Research Projects Agency (DARPA).

B. OVERVIEW OF SAAM

Data networks today (i.e., the Internet) are built by using sophisticated stand-alone routers. Each router is involved in routing and network management tasks in addition to data forwarding. As networks grow to handle more and increasingly diverse data, more processing will be required for each router. A legacy router can easily be a performance bottleneck due to a lack of processing power [2].

SAAM is a networking architecture that addresses this problem and provides an efficient solution to integrated services. Unlike the distributed architecture of current networks, SAAM relieves most of routing and network management tasks from individual routers. Instead, SAAM deploys a small number of dedicated servers to perform these tasks on behalf of the routers (see Figure 1.1) [2].

SAAM is envisioned to be the network environment where routing, resource reservation, network management, accounting and security can be integrated [2]. Concentrating the network management and control tasks to a small number of servers rather than distributing them to all devices in the network, SAAM facilitates faster development and deployment of new services. Additionally, SAAM is completely compatible with legacy networks. It supports existing inter-domain routing protocols. Therefore, the deployment of SAAM can be incremental, within one autonomous system at a time.

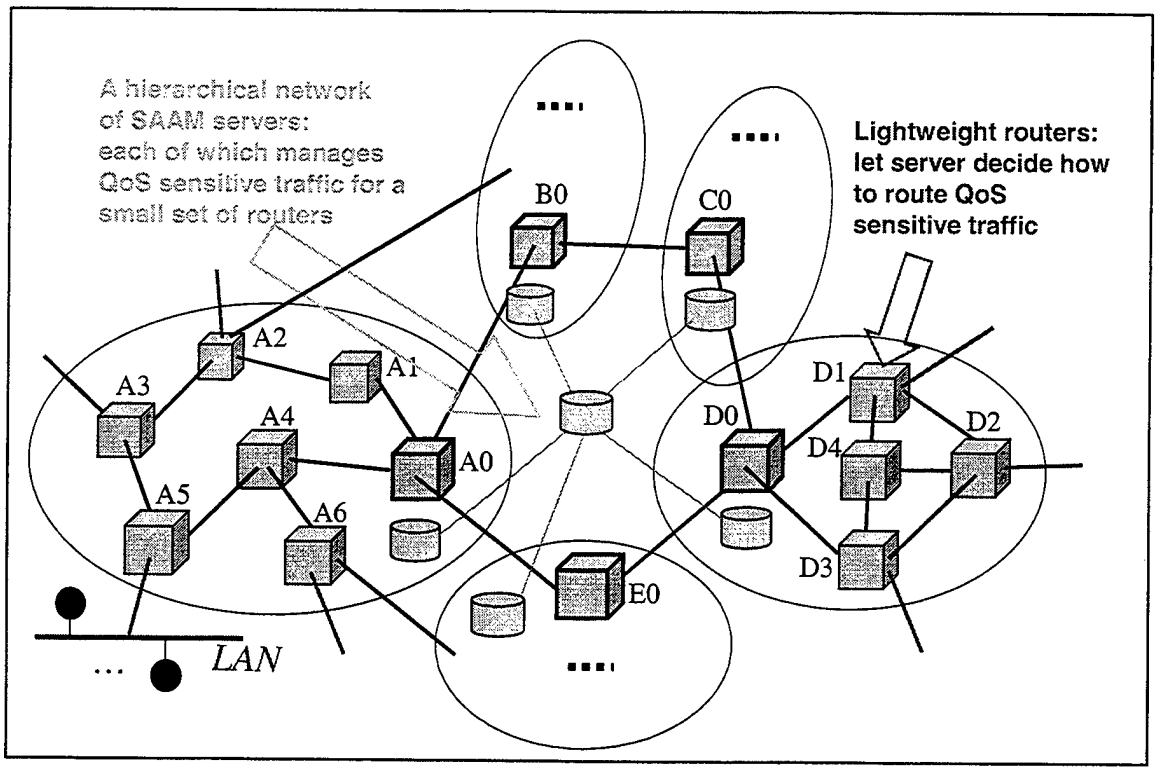


Figure 1.1 Sample SAAM Network Architecture.

1. The Logical Model Of SAAM

The server deployment is the most noticeable character of SAAM architecture. SAAM servers relieve routers from most network control tasks. With the relief provided from the server, the routers used by SAAM can dedicate their resources to network traffic more efficiently than legacy routers.

Each SAAM server maintains a path information base (PIB) for its region (see Figure 1.2). Specifically, the server determines the paths that can be used to route flows and maintains up-to-date performance parameters for these paths by processing Link State Advertisement (LSA) packets sent by routers in its region. Using the PIB, the server can intelligently perform network functions such as QoS routing and re-routing of real-time flows, which are crucial to integrated services. In the logical model of SAAM, each

router is a client of a single SAAM server. The router does not participate in QoS routing. It updates its flow-based routing table with route data passed from the SAAM server [2].

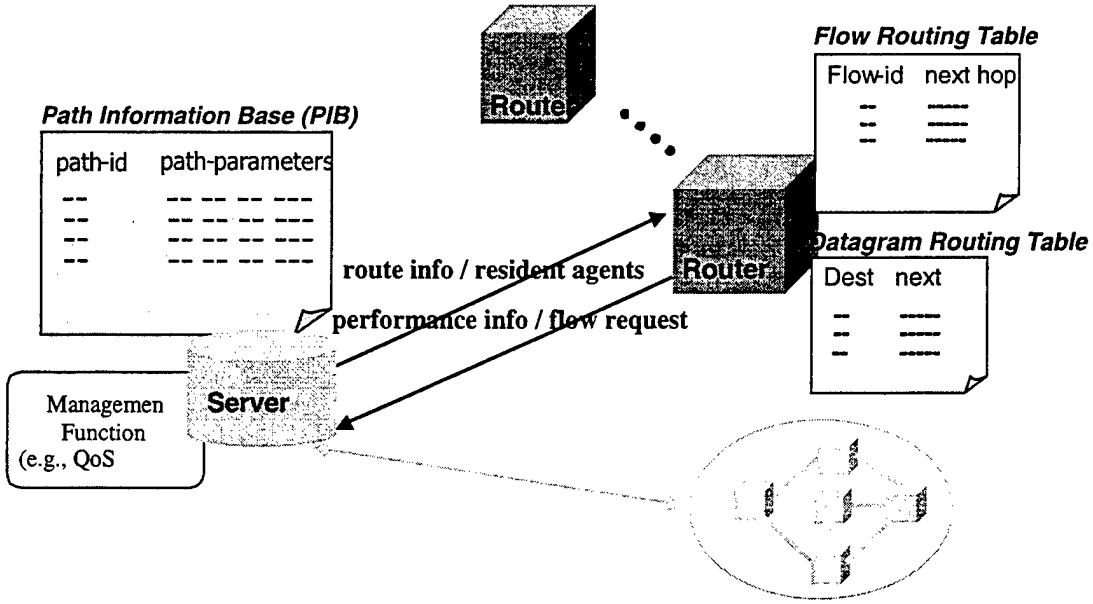


Figure 1.2 Logical Model Of SAAM.

2. Hierarchical Organization of SAAM Architecture

A city traffic monitoring system presents a good example for one to understand SAAM's hierarchical organization of servers. Vehicles or helicopters can be used for traffic monitoring in a city. A motorist or a policeman deployed along a highway or at an intersection can only observe traffic within a small perimeter. On the other hand, an observer on a helicopter can observe the state of traffic of a larger physical area. By coordinating traffic state information of the larger area, the observer can provide more reliable and more valuable information than a number of motorists operating individually. The superior information extracted by utilization of a helicopter can also be used to take measures to prevent probable traffic congestions. Turning back to the realm of

networking, stand-alone routers of today's networks react slowly to traffic fluctuations, like individual motorists. On the other hand, each SAAM server functions like a helicopter based observer. The server always has region-wide information to support its decision-making. With this information, the server can allocate network resources intelligently and adapt to changes in network conditions promptly.

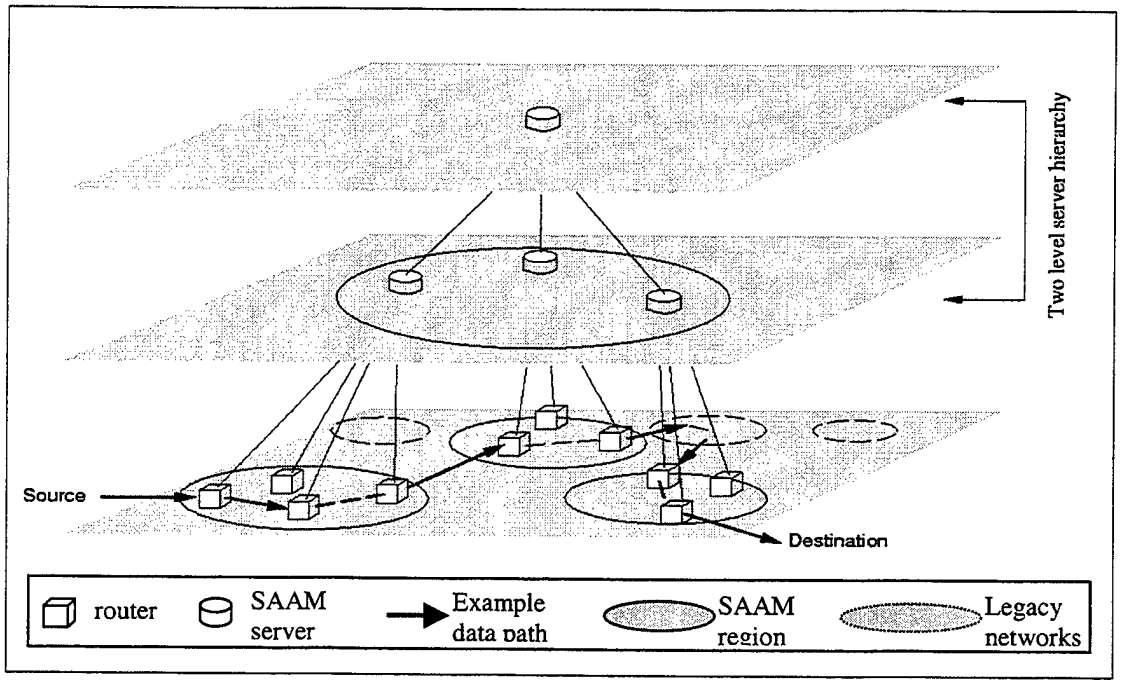


Figure 1.3 Hierarchical Organization Of SAAM Servers.

For scalability, SAAM organizes its servers in a hierarchy (see Figure 1.3). At the lowest level of the hierarchy, SAAM partitions the network into segments called SAAM regions. A SAAM server is deployed for each SAAM region. Each SAAM server is responsible for collecting network state information from routers in its own region. A subset of routers in each region are designated *border gateways* to handle traffic across multiple regions [2]. The regional SAAM server summarizes its regional information and passes the summary to a higher-level (parent) server. The higher-level SAAM server

integrates data from different regions and uses the information to perform inter-regional routing.

C. SCOPE OF THIS THESIS

The goal of this thesis is to develop a routing protocol for establishing signaling channels between a SAAM server and its routers. The efficiency of SAAM depends heavily on these signaling channels being fast, reliable, and cheap. The cost of a signaling channel is measured by the amount of processing and communication overhead it imposes on routers. The routing protocol is required to detect and respond to network topology changes within a fraction of a second. It must be robust, efficient and scalable. This thesis describes such a routing protocol.

D. MAJOR CONTRIBUTIONS OF THIS THESIS

The routing protocol developed in this thesis meets all specific signaling channel requirements of SAAM. The protocol has the potential to be used in any multi-service network architecture (e.g., bandwidth broker or ATM PNNI) that is similar to SAAM.

E. ORGANIZATION OF THIS THESIS

The remainder of this thesis is organized into the following chapters:

- Chapter II: Related Topics. States signaling channel tasks that are to be handled in SAAM architecture. Explains the shortcomings of routing protocols that are readily available today and why they can't be used in SAAM networks.
- Chapter III: SAAM Signaling Channel Configuration Protocol Design.

Describes the routing protocol designed for SAAM architecture.

- Chapter IV: SAAM Signaling Channel Configuration Protocol Implementation. Explains the integration of the developed protocol and the existing SAAM emulation software.
- Chapter V: Test Of SCCP. Describes the application of SCCP to a sample SAAM network and discusses the results.
- Chapter VI: Conclusions. Summarizes the thesis study and interprets the test results obtained from a SAAM test-bed. Also states the future work that needs to be carried out.

THIS PAGE INTENTIONALLY LEFT BLANK

II. RELATED TOPICS

In this chapter, motivations that have led to the development of a special routing protocol for configuration of SAAM signaling channels are introduced. In a SAAM region, a set of signaling channels must be established so the server and routers can exchange vital management and signaling messages. First, the general properties of routing protocols are discussed. Next, the distinct properties of SAAM signaling traffic that stipulates the need of a new routing protocol are explained. Then interior and multicast routing protocols are described since these protocols address similar requirements. The reasons for not using these protocols to configure SAAM signaling channels are presented.

A. ROUTING PROTOCOLS

A routing protocol is a distributed application used by a network to coordinate the task of packet forwarding at different routers. This coordination is required for the network to adapt to changes in topology and input load without significant performance degradations.

Traffic in a network can be divided into two major groups: *signaling traffic* and *user traffic*. *User traffic* consists of all traffic that is generated by user applications. On the other hand, s*ignaling traffic* is used to configure and manage the network in order to keep the network functioning properly. While the user traffic is discernible by the users, signaling traffic runs in the background and it is transparent to the users.

The network forwards *user traffic* using the services provided by *routed* and *routing protocols*. IP and IPX are examples of routed protocols. These protocols provide

enough information in the network header of packets to enable a router to determine the appropriate routing table to use and classify the packets accordingly. *Routing protocols* establish and maintain one or more routing tables at each router. Each entry of a routing table typically consists of two fields: packet classification and next hop. It dictates that all packets with the same classification (e.g., going to the same destination) be forwarded to the same next hop. Examples of routing protocols are Routing Information Protocol (RIP), Open Shortest Path First (OSPF), Interior Gateway Routing Protocol (IGRP), etc. [6]. In summary, the routed protocols provide services to the user traffic by utilizing the routing tables established by the routing protocols.

The *signaling protocols* are a special subclass of routing protocols. They establish and maintain the routing tables specifically for signaling traffic. Paths between routers that are designated for use of signaling traffic are called *signaling channels*.

A routing protocol must fulfill the following requirements.

1. Robustness

The worst thing that a router can do is to misroute a packet. Misrouted packets will never reach their destination and will waste network bandwidth. Moreover, the routing tables that contain incorrect entries may develop formation of loops as well as superfluous oscillation of traffic load between nodes. These are detrimental problems that may render a network unusable. A robust routing protocol should protect itself from these problems by periodically running consistency tests. Checksums and sequence numbers should also be used for the signaling messages of a routing protocol to ensure the integrity of these messages [5].

2. Manageable Signaling Overhead

Routing protocols establish the routing tables by exchanging signaling messages between nodes [5]. These signaling packets share bandwidth, queue space and CPU time with user traffic. The more a routing protocol uses these resources, the less they are available to user traffic. Routing protocols that control the overhead resulting from circulation of signaling messages to a manageable level are preferred since they do not disrupt user traffic.

3. Path Optimization

It is always desirable to route a packet from source to destination using the best available path. Determination of the best path depends on what is considered the best. The highest priority may be assigned to the path with maximum bandwidth, minimum delay, maximum safety, etc. [5]. When multiple paths are feasible, a good routing protocol should select the best path according to a given metric.

4. Manageable Routing Table Size

Some routing protocols require neighbors to exchange their entire routing tables. Maintaining a large routing table imposes significant processing overhead upon a router. Routing protocols that use manageable routing table size are preferred [5].

No routing protocol can be the best with respect to all the requirements. For this reason, trade-offs are inevitable between different requirements. It is the protocol designer's responsibility to determine which requirements are more important for a given situation.

B. DISTINCT PROPERTIES OF SAAM ARCHITECTURE RELATED TO SIGNALING CHANNELS

Understanding the particular properties of SAAM signaling traffic is essential for determination of the issues that need to be addressed by the new routing protocol for SAAM signaling traffic. The following describes the main issues related to signaling traffic in SAAM.

1. Asymmetric Flows of Signaling Traffic

In SAAM, signaling traffic is exchanged between the server and its routers. The server needs to talk to all routers in its region. The signaling traffic originating from the server and destined to routers travels from a single source to multiple destinations. Every router also needs to talk to the server in its region. The signaling traffic destined to the server travels from multiple sources to a single destination. Therefore, the signaling traffic flows are asymmetric. In particular, the traffic is unevenly distributed, heavier on the routers closer to the server. That asymmetry is one of the most distinctive properties of SAAM architecture.

2. Pro-Active Response to Topological Changes

In SAAM, topology changes must be determined and handled as they occur to support guaranteed services. Local detection of topological changes and hop-by-hop dissemination of knowledge of the changes is not optimal for SAAM architecture. A reactive method of updating routing tables takes too long for real-time traffic. In SAAM, real-time reconfiguration of signaling channels is mandatory in order not to affect traffic requiring hard QoS guarantees. User traffic should not experience any noticeable degradation of services while signaling channels are re-established.

C. INTERIOR ROUTING PROTOCOLS

An *Autonomous System* (AS) is defined as a set of routers and end-systems that are administered by a single authority [5]. Interior routing protocols determine how packets should be routed inside an AS. A SAAM region can be considered as an autonomous system under the administration of the server. Therefore, understanding of interior routing protocols may contribute to development of the protocol that configures signaling channels of SAAM.

Interior routing protocols use either distance vector or link-state algorithms. Next, each of these algorithms is examined for its suitability for SAAM signaling channels.

1. Distance-Vector Algorithm

In a distance-vector algorithm, each node stores < destination, cost > information for all other nodes and exchange this information with its neighbors. The distance-vector algorithm assumes that each node knows the link cost to each of its directly connected neighbors [3]. Initially, a router assigns an infinite cost to non-neighbor routers. Routers periodically send their distance vectors to all their neighbors. When a router receives a distance-vector from a neighbor, it re-evaluates the cost to reach each destination by summing the cost advertised by the neighbor and the cost of reaching that neighbor. If the total cost is less than what is stored at its distance vector, the router updates the distance vector with the new cost and uses that neighbor as the next-hop for the destination.

The distance-vector algorithm is distributed, asynchronous and iterative. The algorithm is distributed because each node distributes its distance vector information to neighbors. It is iterative because each node recalculates the cost to reach destination

nodes until no more distance vectors are exchanged between nodes. It is asynchronous because, exchanges and calculations don't have to be at the same time at all the nodes [4].

The distance-vector algorithm suffers from the problem of slow convergence upon topological changes. Another major disadvantage of distance-vector algorithm is the *count to infinity problem* that happens as a result of incorrect information exchange between nodes. When routers update their cost of reaching a destination using the incorrect information, cost of reaching to that destination increases continuously unless some preventive action is taken. The packets that are sent to a destination whose cost is continuously increasing will circle between the nodes many times without reaching the destination. That will cause congestion and decrease throughput. A method called *split horizon* has been developed to prevent the count to infinity problem. The split horizon approach stipulates that a cost update should not be sent back to the node from which the information triggering the update has been received. Even though split horizon prevents two neighbor nodes from counting to infinity, it is ineffective for more than two nodes. So it cannot guarantee problem-free operation in all cases. The most popular distance-vector algorithm based routing protocol is the Routing Information Protocol (RIP), which is discussed below.

Routing Information Protocol (RIP)

RIP uses hop count as the metric to measure the goodness of a route. It assigns a cost of 1 to directly connected neighbors. On the other hand, an unreachable destination is assigned a cost of 16. For RIP, a route with the minimum number of hops is the best regardless of other issues. RIP routers exchanges distance vectors every 30 seconds. A destination is concluded to be unreachable if it does not advertise its distance vector to a

neighbor for six consecutive advertisement intervals (6*30 = 180 seconds). In addition to regular distance-vector exchanges at every 30 seconds, a router sends its distance-vector when a distance-vector from a neighbor causes the router to update its table. RIP uses *split horizon with poisonous reverse* to prevent the count to infinity problem. While in the split horizon approach, a router does not advertise the cost of a destination to the neighbor who just advertised it, in the split horizon with poisonous reverse approach the router advertises infinite cost for that destination. RIP is applicable to small networks where routes have fewer than 16 hops. It is simple to manage and configure. However, it is inadequate in supporting multiple metrics and cannot overcome failures quickly [5].

2. Link State Algorithm

The link state algorithm assumes that each router is capable of determining the state of its links to neighbors (up or down) and their costs [3]. Routers periodically create *Link State Packets* (LSPs) that describe the current state of their links. An LSP contains the *routers ID*, the *neighbor's ID*, the *cost* of the link to the neighbor, a *sequence number* and a *time-to-live* value. Every router sends its LSPs to every other router in the topology. LSPs are distributed via flooding. A router retransmits a received LSP to all its neighbors except the one that has forwarded the LSP. An LSP is kept at a router for a maximum of time-to-live amount of time. Upon expiration of that time, the associated LSP is deleted. The sequence numbers helps a router determine whether a LSP is fresh. Those LSPs with an invalid sequence number are dropped.

Since a router receives LSPs from all the other routers in the topology, it can have an overall map of the network. A router calculates the best path to every other router in the topology, constructing a spanning tree with itself being the root. *Dijkstra's shortest*

path algorithm can be used to compute the shortest-path spanning tree. The algorithm iteratively searches for the best path from a set of nodes that are already part of the spanning tree to the nodes that are not part of the tree yet [3].

Open Shortest Path First (OSPF)

As the size of the Internet increases, the limitations of RIP become more evident, which has eroded the protocol's popularity. Today, Open Shortest Path First (OSPF) is the preferred interior routing protocol for TCP/IP based networks [1].

OSPF partitions ASs into areas. As a result of the extra level of hierarchy introduced by areas, a router does not need to know all the destinations in the AS. It rather needs to know the routers in its area. For routers outside its area the router needs to know the gateway router leading to that area. Dividing an AS into areas reduces not only the amount of data stored at each router but also the amount of information that must be exchanged between routers. The reduction facilitates efficiency and scalability.

In OSPF, a router that receives a link update message authenticates the sender. Authentication brings robustness to OSPF, preventing the network from being shut down by a misconfigured host. A misconfigured host may send LSPs with incorrect information. Such LSPs may cause improper updates of routing tables at the receiving routers. A misconfigured routing table will route the network traffic to inappropriate destinations, which is totally undesirable. So authentication of messages ensures formation of consistent routing tables, which is vital for proper functioning of the network.

OSPF exchanges signaling messages much less often than RIP. As a result, it imposes lower overhead upon the network. Additionally it can support up to five different

types of metric. For each metric a different shortest path tree is calculated and a separate routing table is formed.

OSPF employs *HELLO* messages to detect the state of its neighbors. If a router does not acknowledge the *HELLO* messages, its neighbor or neighbors will declare its failure and send an LSP to inform others of the failure. A router re-calculates the distance to each destination upon receiving this LSP.

The main disadvantage of OSPF is the high overhead of initial flooding of LSPs. The amount of the overhead is directly proportional to the connectivity of the network [1].

3. Suitability of Interior Routing Protocols for SAAM

In SAAM, topological changes must be determined and handled as they occur to support guaranteed real-time services. Interior routing protocols are based on local detection of topological changes and hop-by-hop dissemination of knowledge of the changes, which are too slow for SAAM architecture. Moreover, these protocols assume a totally distributed network architecture where each router must construct the view of the entire region. They seem overly complex for SAAM where only the server needs to have such a view.

D. MULTICAST ROUTING PROTOCOLS

Multicasting is the process of sending a packet from a source to multiple destinations with a single transmit operation [4]. In SAAM, since the server needs to have the capability of communicating with every router, the profile of server-to-router communication is similar to multicasting. Because of this similarity, multicast routing

protocols are examined for their applicability to configuration of SAAM signaling channels.

Since multicasting requires routing from a single source to multiple receivers, the goal of multicast routing protocols is to determine the tree of links that connects all receiving nodes to the sender. This is basically achieved either by *group-shared tree approach* or by *source-based tree approach* [4]. Before their suitability or unsuitability to SAAM is asserted, both approaches are briefly described next.

1. Group-Shared Tree Approach

The group-shared tree approach constructs a single routing tree to be used by all nodes that are members of the multicast group. In figure 2.1, a network topology where the group-shared tree is marked by dark lines is shown. A group–shared tree can be used for bi-directional traffic flow.

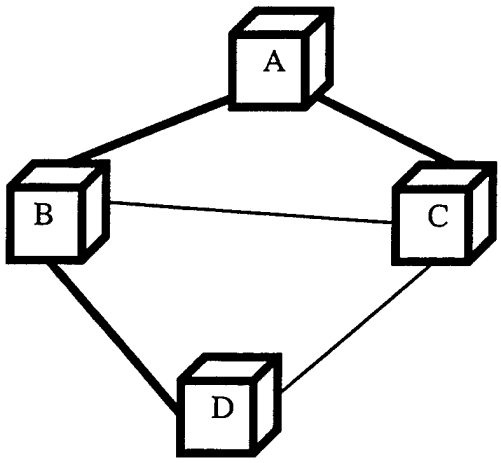


Figure 2.1 A Sample Group-Shared Tree.

A common method to determine the group-shared multicast tree uses the notion of a *center* node. A router sends a JOIN message to the center node to join the multicast tree. The *JOIN* message is forwarded using unicast toward the center until it reaches the center node or a router that already belongs to the tree. At that time, the join process is competed, and the route traveled by the *JOIN* message forms a new branch of the multicast tree [4].

The Core-Based Tree Multicast Routing Protocol that has been put into practice in the Internet implements the group–shared tree approach. The details are presented below.

Core Based Tree Multicast Routing Protocol

The core based tree (CBT) multicast routing protocol is specified by RFC 2201 and RFC 2189. It constructs a bi-directional, group-shared tree with a core (center) node. Routers join the tree by sending a *JOIN_REQUEST* message to the center. The core or the first router that receives the join request message acknowledges the request with a *JOIN_ACK* message. The multicast tree is maintained by exchanging *ECHO_REQUEST* and *ECHO_REPLY* messages between the downstream and upstream routers [4].

2. Source Based Tree Approach

The source-based tree approach constructs a separate tree for each sender in the multicast group. Unlike the group-shared tree approach, some links of the trees may be used in one direction only. A sample source-based tree is shown in figure 2.2, in which the source-based tree rooted at router B is represented by normal arrows while source-based tree rooted at routers D is represented by dashed arrows.

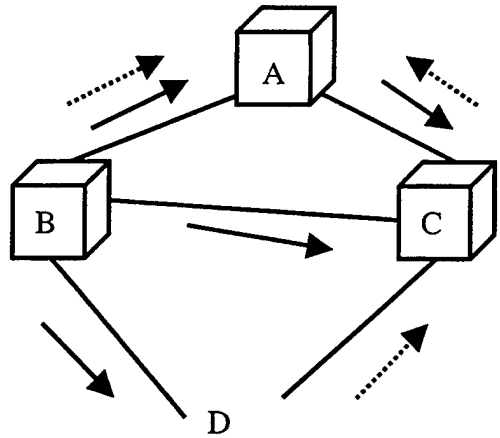


Figure 2.2 A Sample Source-Based Tree.

Two currently standardized Internet multicast routing protocols, Distance Vector Multicast Routing Protocol (DVMRP) and Multicast Open Shortest Path First (MOSPF) are described below.

a) Distance Vector Multicast Routing Protocol (DVMRP)

DVMRP implements source-based trees with reverse path forwarding, pruning and grafting. It uses the distance vector algorithm to determine the next hop for each destination address. In DVMRP routers monitors their dependent downstream routers for possible pruning as well. A prune message is initiated by a leaf router that no longer has any multicast group member on its directly attached subnetworks. When a router receives a prune message from all its dependent downstream routers, it sends a prune message to the upstream router. Prune messages carry a *life-time* field with a default value of two hours. This is the time during which the branch will remain pruned before being restored automatically. Graft messages may be sent from routers to their upstream routers to reactivate previously pruned branches [4].

b) Multicast Open Shortest Path First Protocol (MOSPF)

MOSPF can be used in autonomous systems that use OSPF for unicast interior routing. MOSPF extends OSPF by having routers add their multicast group membership information to the link state packets. MOSPF routers build the routing table for multicast routing as well as unicast routing. Using MOSPF routers can build source-based, pruned shortest path multicast routing tables [4].

3. Why Multicast Routing Protocols Can't Be Used For SAAM

Multicast protocols arrange possible receivers into groups and assign each group a unique address. A source appends the group address to each packet that it sends to the group. There is no protocol that allows the source to determine the addresses of the receivers in a multicast group [4]. Basically the sender has no idea about the number and identity of its receivers. Additionally any source can send a multicast packet to all the receivers.

In SAAM, only the server needs to communicate with every router. The server sends different signaling information to different routers. That is concurrent unicast rather than multicast routing.

The core-based tree approach seems the closest match to deal with signaling traffic of SAAM. But in that approach, a new router joins the tree simply by discovering an existing member of the tree. In contrast, in SAAM a router's join message is required to arrive at the server so that the server knows which next hop to use for that router. Another problem is the use of a single group address for all the group members. SAAM signaling messages are usually destined to one destination. The server and routers need to

know the unique address of each other for individual communication as well as for security reasons. This issue is not addressed by multicast routing protocols.

For these reasons multicast routing protocols can't be used to configure the SAAM signaling channels either. Therefore, a routing protocol that addresses the specific requirements of SAAM signaling traffic needs to be developed.

III. SAAM SIGNALING CHANNEL CONFIGURATION PROTOCOL DESIGN

It follows from Chapter II that existing routing protocols do not adequately address the needs of signaling traffic of SAAM. In this chapter, a new routing protocol especially designed for SAAM signaling traffic, called Signaling Channel Configuration Protocol (SCCP), is described. It should be noted that SCCP configures the channels to carry the signaling traffic in SAAM. The protocol itself does not carry signaling traffic. The rest of the chapter is organized as follows. First, SCCP is briefly introduced. Then, how SCCP achieves timely discovery and response to topological changes is explained. Major issues affecting the design of SCCP are discussed next, followed by a detailed description of the SCCP algorithm. SCCP has been applied to a sample topology. The results are discussed in the last section.

A. INTRODUCTION TO SIGNALING CHANNELS CONFIGURATION PROTOCOL

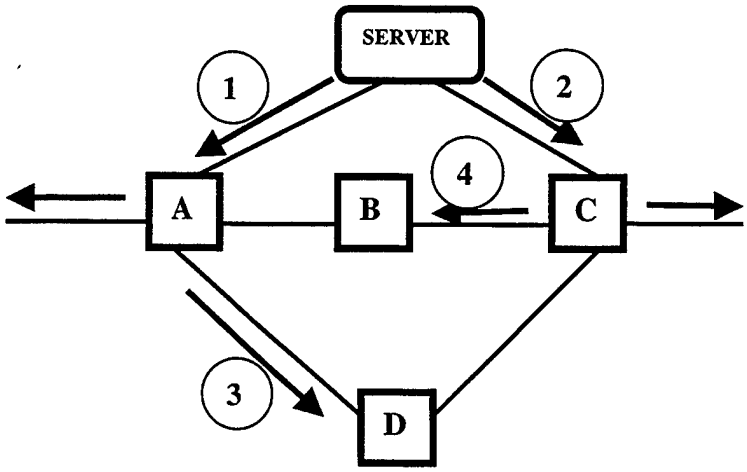


Figure 3.1 SAAM Topology Displaying Router-Bound Signaling Channels.

A sample SAAM topology is shown in figure 3.1. The server requires ready-to-use channels that go to all routers in the SAAM region. These channels are required to carry the signaling traffic originating from the server and destined to routers. They are called *router-bound signaling channels.* In the sample topology, the server uses channel 1 to send signaling messages to router A, channel 2 to router C, combination of channels 1 and 3 to router D, and combination of channels 2 and 4 to router B.

On the other hand, each router needs to know the next hop to forward its signaling traffic to the server. In figure 3.2, the topology of figure 3.1 is used to show the signaling channels from routers to the server. This time, router A uses channel 1, router C channel 2, router D a combination of channel 1 and 3, and router B a combination of channels 2 and 4 to send signaling messages to the server. These channels are called *server-bound signaling channels.*

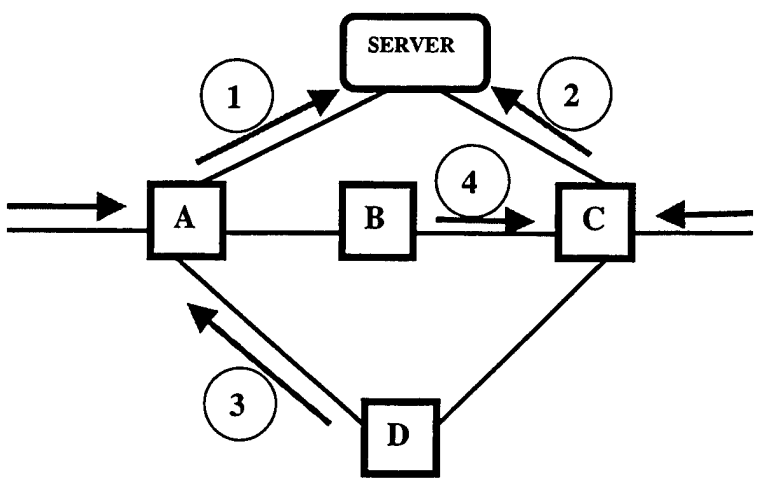


Figure 3.2 SAAM Topology Displaying Server-Bound Signaling Channels.

The shape of channels formed by server-bound and router-bound signaling channels connotes a tree structure. This tree formation has motivated the use of a spanning tree algorithm as the basis for SCCP.

1. The Spanning Tree Approach

The first spanning tree algorithm was developed by Radia Perlman at Digital for bridge routing. To avoid loops, a set of bridges may agree upon a spanning tree for a particular extended LAN[1]. If an extended LAN is thought as being connected by a graph that possibly has loops, then a spanning tree is a sub-graph of that graph that covers all the vertices (LANs) but contains no loops. The bridges form a spanning tree by deactivating some of their ports. They can reconfigure themselves into a new spanning tree should some hardware failure occur [3].

The spanning tree approach used by LAN bridges addresses several issues that are also applicable to SAAM architecture. For example, the algorithm establishes a spanning tree that is rooted at the *root bridge* and covers all the bridges. It also determines those links that should be included for establishing complete reachability. This approach is well suited for establishing the connectivity between the server and all the routers in a SAAM region. For this reason, the spanning tree algorithm has been taken as the starting point in design of the SCCP.

The spanning tree algorithm employed by LAN bridges first elects the root of the spanning tree. Then, all bridges connected to a given LAN elect a single *designated*

[1] An **extended LAN** is a collection of LANs connected by bridges[4].

bridge that will be responsible for forwarding frames toward the *root bridge*. Each LAN's designated bridge is the one that is closest to the root bridge [3].

In SAAM architecture, the server acts as a control center. It is logically the root of the spanning tree for SAAM signaling channels. Pre-defining the root rather than determining it with elaborate message exchange between candidate nodes enhances the efficiency of the spanning tree algorithm implemented in SCCP.

The spanning tree algorithm may be used to configure uni-directional signaling channels that originate from the server. However, SAAM requires bi-directional signaling channels. As much as the server needs to send management and control information to every router, each router needs to send its link state statistics and forward flow requests to the server. Therefore, the spanning tree algorithm needs to be extended for SCCP. This extension is essential for the bi-directional flow of SAAM signaling traffic.

2. How To Make A Two-Way Spanning Tree

In SCCP, the first task is the construction of a spanning tree rooted at the server and covering all the routers by using the spanning tree algorithm. This task can be accomplished by using a message that advertises the server to the routers. This message is called *"Downward Configuration Message (DCM)"* in SCCP.

The DCM mechanism is devised in such a way that upon receiving the first DCM a router designates the input port (interface) for that DCM to be part of the server-bound signaling channel and generates the routing table entry to use for server-bound signaling[2] traffic. Then the router forwards a copy of the DCM to all its neighbors except the one

[2] At a router, signaling traffic is classified based on the flow id carried by each packet.

where the first DCM has come from. The router will drop all subsequent DCMs that it receives.

a) Router-Bound Signaling Channel Problem

Even though the server-bound channels are established via DCMs, the server cannot communicate with the routers since it does not know anything about the routers yet. In order for the server to attain this capability, two requirements must be fulfilled. First, the server must have the necessary entries in its routing table for all routers in its SAAM region. Second, in order for server generated signaling messages to reach a destination router, all routers on the branch of the spanning tree that connects the server with the destination router must have an entry for that destination in their routing tables. These two requirements may be accomplished by a number of different ways.

It might be possible to manually configure each router. But that is out of the question for a dynamic network like SAAM.

Another option is to use one of the existing interior routing protocols. But, it follows from Chapter II that these protocols do not provide the optimal solution for the SAAM environment.

One other way might be to have each router reply to the server by a message upon receipt of a DCM. Such a reply message will pass through all the *ancestor*[3] routers before arriving at the server. Each visited ancestor router can then update its routing table to include an entry for forwarding signaling messages destined to the replying router. Indeed a reply message is basically required for each router to reveal itself to all its ancestors including the server. Moreover, all the reply messages carry the

[3] For definitions of ***root, parent, child, ancestor, descendant,*** *etc.,* see Appendix A.

same header information except the source address. The redundant bits in the reply messages introduces large amount of overhead that wastes bandwidth and CPU time. Additionally, the routers that have many *descendant* routers must process many reply messages, which reduce the throughput of those routers further. For example, in figure 3.3, router A processes as many reply messages as the server. Consequently the use of reply messages may introduce large overhead for configuring the router-bound signaling channels.

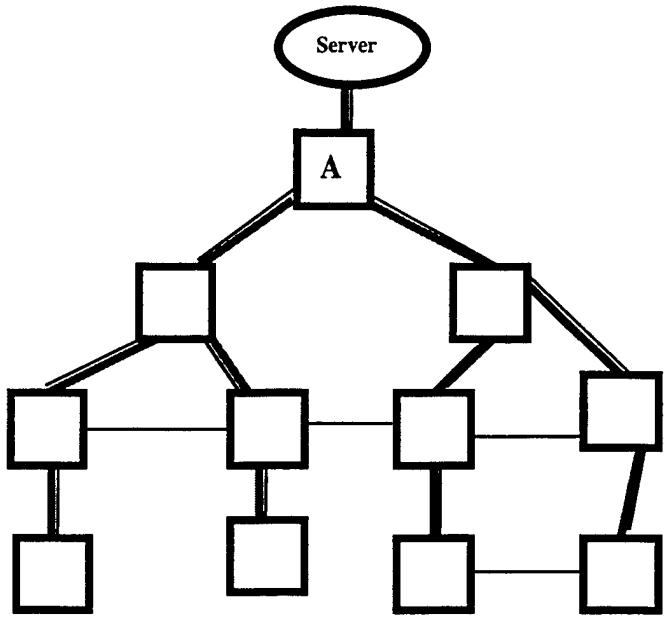


Figure 3.3 SAAM Topology With Configured Signaling Channels (Thick Lines).

In view of the fact that none of these approaches is optimal for SAAM, another approach that overcomes the shortcomings of the aforementioned approaches has been developed.

b) Designed Solution

The designed solution still uses reply messages to configure the router-bound signaling channels, but in a different manner. The reply message used in SCCP is

called *"Upward Configuration Message (UCM)"*. The details of the approach are described next.

In this approach, the routers aggregate UCMs from their child routers instead of forwarding them immediately towards the server. As a consequence of this aggregation, a single UCM may be sufficient to configure the router-bound signaling channels for the parent and all the descendant routers. The aggregation is achieved by using one extra message, two timers and two buffers.

The extra message used to achieve UCM aggregation is called *"Parent Notification (PN)"* message in SCCP. It is sent from a child router to its parent router right after the child processes the DCM from the parent. Upon receipt of a PN message the parent learns the existence of the child router. PN messages allow a router to know how many UCMs to expect and where they should originate. After all expected UCMs have been received and processed, the router will stop the aggregation process and forward a UCM on behalf the entire branch. Each PN message is basically required to carry the id of the child router. Consequently it is small in size and does not introduce considerable overhead.

The two *timers* running on each router are intended to prevent infinite wait for a UCM from a child router. Both timers are started right after the router forwards the DCM to its neighbors. The first of the two timers is called *local timer*. It is cancelled when the router receives the first PN message. When this timer expires, the router will declare itself as a leaf node and send a UCM to its parent. The other timer is called *global timer*. It is cancelled when the router receives UCMs from all known child routers. Upon cancellation the router sends a UCM to its parent. Conversely if the global timer expires

before receipt of all UCMs from child routers, the router will stop the aggregation process and send a UCM to its parent, assuming some of its children have failed. This will prevent infinite wait for a UCM from a child router.

The two buffers are employed to store the ids of child routers temporarily for the UCM aggregation process. The first buffer is called *noticationBuffer* and it is used for storing the id of every router from which a PN message has been received. The other buffer is called *reachabilityBuffer.* It is used for storing ids of descendant routers that are learned from the UCMs.

B. HOW SCCP HANDLES TOPOLOGY CHANGES AS THEY OCCUR

SAAM stipulates pro-active response to topological changes. So SCCP is obliged to actively detect and respond to changes of topology. This requirement is fulfilled by a soft state approach where signaling channels are completely reconfigured periodically or when there is a need. Instead of keeping the state of every channel in the topology and waiting for a change, SCCP partitions time into fixed-length intervals called *refresh intervals* and establishes new signaling channels at the beginning of each refresh interval. The frequency of signaling channel configuration (i.e., the length of refresh interval) is an important design parameter. While a shorter refresh interval is favorable to accommodate topological changes, the overhead of SCCP is also larger with a shorter refresh interval. So employing an appropriate refresh interval is crucial to the effectiveness of SCCP. The determination of an optimum refresh interval is beyond the scope of this thesis.

SCCP make use of a *sequence number* field inside each message that it uses. The sequence number is intended to uniquely identify the configuration cycle in which the message is created. As a consequence of periodical reconfiguration, messages of previous

cycles may have remained in the network and surfaced in the current cycle. The sequence number allows the routers to recognize and discard these old messages.

C. MAJOR ISSUES THAT HAVE INFLUENCED DESIGN OF SCCP

The approach that assembles a bi-directional spanning tree for configuring SAAM signaling channels has raised some new issues. These issues are related to server-bound signaling channels, router-bound signaling channels and routing tables. Each of these issues is explained separately as follows.

1. Issues Related To Server-Bound Signaling Channels

In a SAAM region multiple servers may be deployed at the same time for fault tolerance reasons [9]. Therefore, SCCP is required to configure the signaling channels for each server. To fulfill this requirement each router must have the capability to discriminate the servers. For this reason, the DCM that is used to advertise a server to routers has to carry a field that is unique to each server.

2. Issues Related To The Router-Bound Signaling Channels

In order to achieve aggregation, a router that sends a UCM should declare all its descendants as well as itself to the parent. So a UCM should carry the ids of all its descendants. SCCP uses *router id* to discriminate routers.

3. Routing Tables Used By SCCP

In SAAM, traffic requiring QoS guarantees is forwarded based on the *flow id*. The routing table used for this kind of traffic is called *FlowRoutingTable*. A FlowRoutingTable has the format shown in figure 3.4. The *flow id* and *service level* are

assigned by the server. The *service level* field specifies the general class of service for the flow. The highest priority is given to service level 0, assigned exclusively to signaling traffic. Other service levels include guaranteed service, differentiated services, and best effort service. The *goodness* field represents the freshness of the entry. The value of goodness field may simply be the sequence number of the latest configuration cycle.

Flow id	**Next Hop Address**	**Service Level**	**Goodness**
k	IP_Address	< i >	Goodness value

Figure 3.4 Flow Routing Table Format.

SAAM architecture reserves flow ids 1 trough 64 for signaling traffic. Since the messages employed by SCCP are critical for the precise functioning of SAAM, these messages are also considered part of the signaling traffic. Accordingly they are assigned flow id too. Different flow ids are assigned to configuration messages used by each server since each server deployed at the same region requires its own signaling channels.

When there are multiple servers in the region, the routers distinguish the servers from one another by a pair of unique flow ids assigned to each of them. SCCP is designed in such a way that, all server-bound signaling traffic for one server uses the same flow id 'k', 'k' being an even integer ranging between 2 and 64, while all router-bound signaling traffic for the same server uses the flow id 'k-1'. Tying all signaling traffic for one server with unique flow ids ensures consistent configuration of signaling channels in the presence of multiple servers.

The *FlowRoutingTable* can be directly used for server-bound signaling channels since it is designed to hold entries of the **< flow id, next hop >** format. On the other hand,

it can't be used for router-bound signaling channels, which need to reach multiple destinations. The router-bound signaling traffic regardless of the destination carries the same flow id ('k-1') and the same source (the server) address. Consequently the format of table entries for router-bound signaling channels should be **< destination, next hop >**. That stipulates a routing table with capability of keeping multiple destinations for the same flow id. For that purpose, a *Router Bound Control Channel Table* (RBCCT) has been developed.

The RBCCT entries are composed of *destination router id*, *next-hop address* and *goodness* members. The flow id of the server ('k-1') is used as key to link a RBCCT to the server whose signaling traffic the RBCCT is forwarding. The RBCCT has the format shown in figure 3.5.

Destination Router Id	**Next Hop Address**	**Goodness**
IP_Address	IP_Address	Goodness value

Figure 3.5 The Format Of RBCCT for Server With Flow Ids 'k' And 'k-1'.

D. SCCP ALGORITHM

The server initiates configuration of signaling channels at the beginning of each refresh interval by sending a DCM to all of its neighboring (directly connected) routers. It increments the sequence number each time it enters a new refresh interval. Each router in the SAAM region stores the sequence number it has extracted from the last valid DCM. As a result when the router receives a DCM with a higher sequence number, it knows the server has started reconfiguration of signaling channels and the DCM is valid. The router will discard any DCM with a sequence number that is less than or equal to the one stored.

SCCP is designed in such a way that, although a router receives a DCM from all of its neighbor routers it processes only the first DCM and drops all the others. In order to build the spanning tree, SCCP assumes the sender of the first DCM is the best next-hop to the server. Since the DCM that experiences the least delay on its way from the server will arrive first, SCCP effectively uses "transit delay" as the metric for route selections. In fact, a link may not have the same delay characteristics in both directions. For example the delay experienced by a server-bound message may be much higher than that experienced by a router-bound message on the same link. In SCCP, link delays are assumed to be symmetric so that a message experiences the same the delay regardless of the direction of travel. An important future work item to investigate is extension of SCCP for configuration of signaling channels where links have asymmetric delays

Upon receiving the first DCM a router designates the sender of this DCM as its parent (next hop) for the server-bound signaling channel and updates its *FlowRoutingTable* accordingly. Then the router sends a PN message to the parent, and sends a DCM to all its neighbors except the parent. Broadcasting DCM that way guarantees that all connected routers in the SAAM region will learn the servers. In the mean time, the router also starts its local and global timers for UCM processing.

When a router receives a PN message from another router, it cancels its local timer if it has not already done so. It places the id of the child router (the sender of the PN message) into the *notificationBuffer*.

A router configures its router-bound signaling channels upon processing UCMs from their child routers. The router sends a UCM when an *event-driven* or *time-driven* condition occurs. The time-driven condition occurs when either the local or global timer

expires. The event-driven condition occurs when the router has received UCMs from all its children registered in the *notificationBuffer*. The first occurring event renders the other ineffective to prevent redundant UCMs. When a UCM triggered event occurs, the router retrieves the router ids from the *reachabilityBuffer* and put them along with its own id into a new UCM. It then sends the UCM to the parent. If the UCM is triggered by receipt of UCMs from its registered children, the router cancels its global timer.

When a router receives a UCM, it first checks the *notificationBuffer* to verify that the UCM is from a registered child. The id of the UCM sender is searched in the *notificationBuffer*. If it is in the buffer, RBCCT is updated for each descendant routers learned from the UCM. These descendant routers are added to the *reachabilityBuffer*. (For the tasks performed at a server and a router see Appendix B and Appendix C respectively).

E. APPLICATION OF SCCP TO A SAMPLE TOPOLOGY

SCCP has been applied to the sample topology shown in figure 3.6. In the sample topology, there is a single server and three routers. It is assumed that configuration traffic originating from the server is assigned flow id '1'.

Configuration of signaling channels is explained for the first cycle (sequence number is 1) as follows. At the beginning of the refresh interval, the server sends a DCM to both router A and router B. It then receives a PN message from both. Since it is the server with no parent, it does not need to create and send any UCM. It only waits for UCMs.

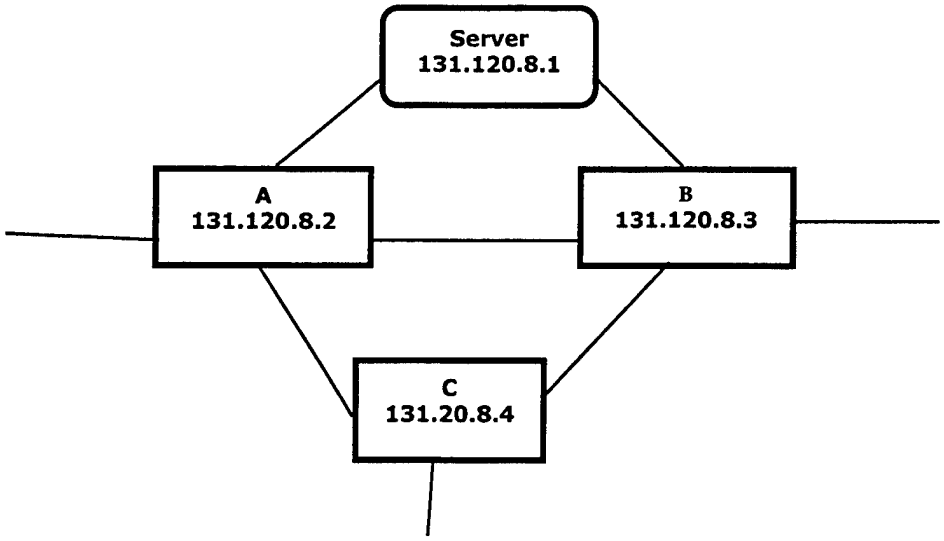


Figure 3.6 SAAM Topology For SCCP Application.

On the other hand, upon processing the DCM received from the server, router A and router B configure their server-bound signaling channels and add a proper entry to their flow routing tables. Then router A sends a fresh copy of DCM to both router B and router C while router B does the same for router A and router C. Router A drops the DCM from B and router B drops the DCM from A. After sending DCM to neighbor routers, both router A and B set their local and global timers. Meanwhile router C receives the DCMs from router A and B. It processes the DCM that arrives first and drops the other one. Assuming the first DCM is from router A. Router C processes that DCM and drops the DCM from router B. It configures the server-bound signaling channel by adding an entry into its *FlowRoutingTable.* At this point, the server-bound signaling channels are configured for all the routers. The routing tables of router A, B and C are shown in figures 3.7, 3.8 and 3.9 respectively.

Flow id	Next Hop Address	Service Level	Goodness
2	131.120.8.1	0	1

Figure 3.7 FlowRoutingTable of Router A After Server-Bound Signaling Channel Is Configured.

Flow id	Next Hop Address	Service Level	Goodness
2	131.120.8.1	0	1

Figure 3.8 FlowRoutingTable of Router B After Server-Bound Signaling Channel Is Configured.

Flow id	Next Hop Address	Service Level	Goodness
2	131.120.8.2	0	1

Figure 3.9 FlowRoutingTable of Router C After Server-Bound Signaling Channel Is Configured.

After processing the first DCM, router C first sends a PN message to router A and then sends a DCM to router B (router B will drop that DCM). Router C also starts its local and global timers for UCM processing.

Upon receiving the PN from router C, router A cancels its local timer and registers the id of router C in its *notificationBuffer*. Meanwhile, the local timers of router C and B will expire since they have not been canceled. The expiration of a local timer triggers the creation of a UCM and the cancellation of the global timer. Since B and C don't have any child router, they only put their own id into their UCM. They forward these UCMs by using the server-bound signaling channels that are already in place.

When router A receives the UCM from router C, it checks whether the id of router C is in the *notificationBuffer*. Finding a record for router C, router A is sure that the UCM is from a valid child. Thus router A updates its RBCCT with an entry for router C and stores the id of router C in the *reachabilityBuffer*. Router C is the only child of router A, so its UCM triggers the even-driven UCM sending and the cancellation of the global timer at router A. Router A retrieves the ids stored in reachabilityBuffer (which currently contains the id of router C only), puts them along with its own id into a new UCM, and forwards the UCM using the server-bound signaling channel.

The server receives a UCM from both router A and B. After these UCMs are processed, signaling channels have been completely configured for the current cycle (see figure 3.10). The server uses the entries in its RBCCT to communicate with any router in the SAAM region.

The router-bound signaling channels for the server, router A, B and C are shown in figures 3.10, 3.11, 3.12 and 3.13 respectively. It should be noted that since router B and router C are the leaves of the spanning tree they have no child. Consequently their RBCCTs are empty.

Destination Router Id	Next Hop Address	Goodness
131.120.8.2	131.120.8.2	1
131.120.8.3	131.120.8.3	1
131.120.8.4	131.120.8.2	1

Figure 3.10 RBCCT Of The Server.

Destination Router Id	Next Hop Address	Goodness
131.120.8.4	131.120.8.4	1

Figure 3.11 RBCCT Of Router A.

Destination Router Id	Next Hop Address	Goodness
-----	-----	-----

Figure 3.12 RBCCT Of Router B.

Destination Router Id	Next Hop Address	Goodness
-----	-----	-----

Figure 3.13 RBCCT Of Router C.

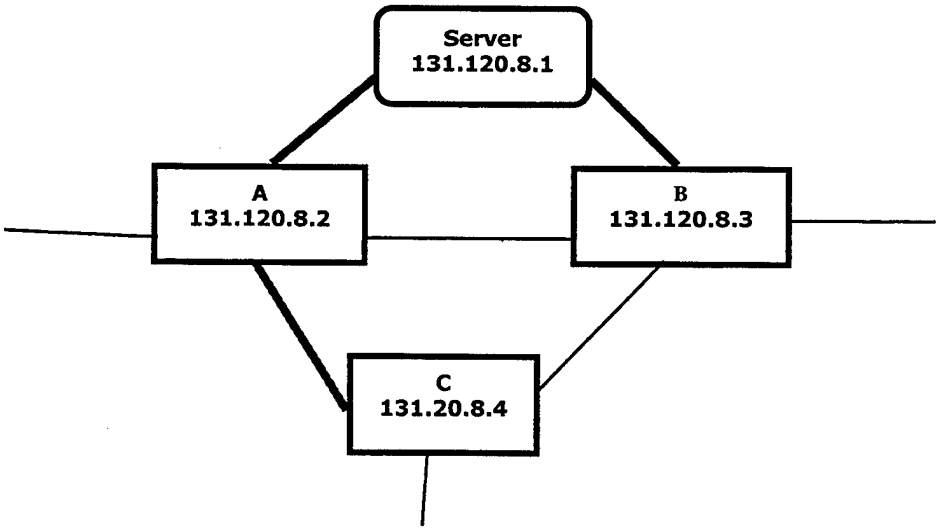


Figure 3.14 Configured Signaling Channels.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SAAM SIGNALLING CHANNEL CONFIGURATION PROTOCOL IMPLEMENTATION

In this chapter the implementation of SCCP on an existing java based SAAM emulation platform is described. First the signaling model used by SAAM emulation software before incorporation of SCCP is described. Then the exact formats of all messages employed by SCCP are defined. Classes that are added to existing software for implementation of SCCP are explained next. Finally, the modifications made to existing classes are described.

A. SAAM SIGNALING MODEL BEFORE IMPLEMANTATION OF SCCP

Before the deployment of SCCP, routers could only send signaling messages to the server via hard-coded server-bound signaling channels. The address of the server and the routing table entries for server-bound signaling channels were sent to the routers out of band from an independent java based application called *DemoStation*. Router-bound signaling traffic was also delivered out of band via dedicated links. This approach is similar to manual configuration of nodes in a network. It is static by nature and requires time-consuming effort from a human administrator each time a topological change occurs.

B. MESSAGES ADDED TO EXISTING SAAM EMULATION SOFTWARE FOR DEPLOYMENT OF SCCCP

As described in chapter III, SCCP uses DCM, UCM and PN messages to dynamically configure the signaling channels. In addition to those messages, two more messages, called *Configuration* and *TimeScale* respectively, have been developed to

make it easier to configure SCCP parameters in the emulation environment (see Appendix D). In SAAM, each signaling message is carried inside standardized packet formats that have been customized for the SAAM emulation environment (see reference 7). The DCM, UCM, PN, Configuration and TimeScale messages are also carried inside those customized packets.

1. Packet Formats Used In SAAM Environment

A packet that carries one or more SAAM signaling messages is called a *SAAM packet*. The current IPv6-based SAAM prototype is emulated at the application layer of an IPv4 network environment. Therefore, all packets of an emulated SAAM test-bed are encapsulated within an IPv4 header on the wire. The resulting IPv4 packets can be classified into two groups: *emulation* and *demo*. Emulation packets carry realistic traffic of a SAAM network and they are used after the SAAM test-bed is initialized. Demo packets carry artificial traffic needed for initialization of the emulation environment. They will no longer be necessary when SAAM is implemented at the network layer.

Emulation packets are transported in the IPv4 environment via TCP port 9001. The format for emulation packets is shown in figure 4.1. A one-byte MAC field is appended to the front of the inner IPv6 packet to simulate the link-layer address of the receiving interface. The inner IPv6 packet could carry either a regular data payload or a SAAM packet. However, only the format of a SAAM packet is shown in figure 4.1. A SAAM packet can be thought as the innermost envelope inside which one or more signaling messages can be placed. It should be noted that all SAAM packets use UDP port 8000 to reach the SAAM control executive of their destination node. The control executive will extract individual signaling messages and process them.

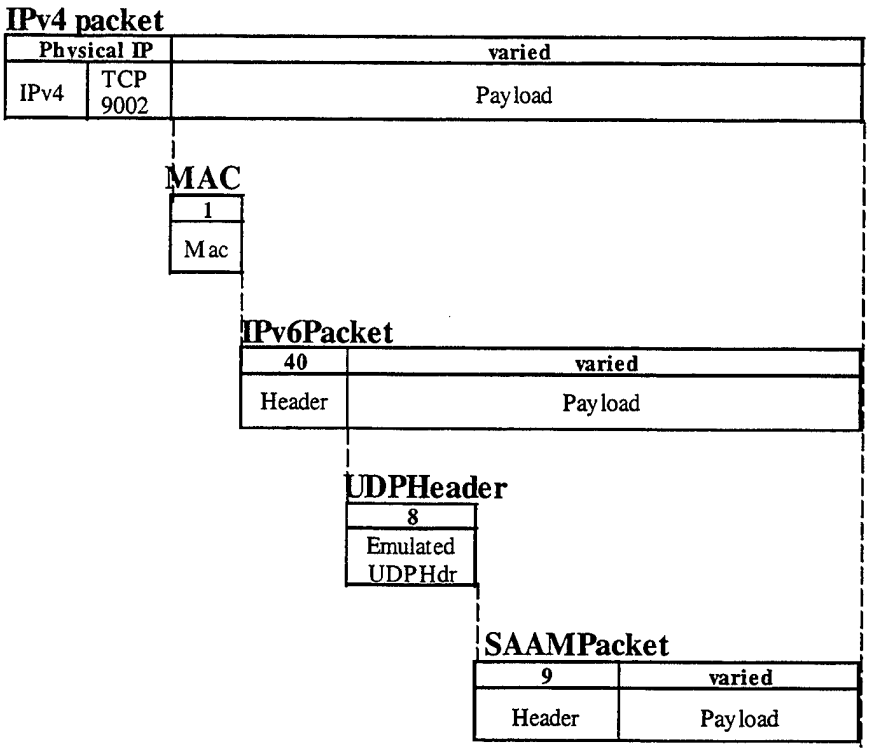


Figure 4.1 Emulation Packet Format.

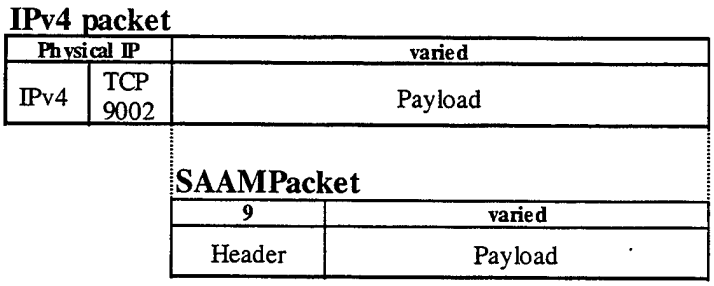


Figure 4.2 Demo Packet Format.

Demo packets (see figure 4.2) are used to set up the topology for an emulated SAAM test-bed. They are transported in the IPv4 world via TCP port 9002. They are typically sent from the DemoStation to a SAAM node. Demo packets bypass normal IPv6 processing. The SAAM packets embedded in them are immediately delivered to the

SAAM control executive without going through all the layers of IPv6 network protocol. Therefore demo packets are like out-of-band traffic.

A SAAM packet comprises of a header and a payload field. The header contains an eight-byte *time-stamp* and a one-byte *number of messages* field. The number-of-messages field declares how many messages are carried inside the SAAM packet. The format of a SAAM packet is shown in figure 4.3. This format reveals the earlier signaling model deployed in the SAAM emulation software. The SAAM emulation software before incorporation of SCCP regarded all messages as an extension of the abstract *Message* class and used one type value of '1' for every message carried inside a SAAM packet. As new capabilities are incorporated into the emulation software, the necessity of creating new messages and differentiating them has emerged. As a result, instead of using the same type value for all messages, the type field in the payload portion of a SAAM Packet is re-defined so that a unique type value is assigned to each signaling message type.

SAAMPacket

9	Varied
Header	Payload

SAAMHeader

8	1
T S	#of messages

Payload portion for Message Format

1	Varied
Type	Message Specific Fields

Figure 4.3 SAAM Packet Format.

2. Integration of Messages Introduced By SCCP to SAAM Packet Format

All the messages used by SCCP (DCM, UCM, PN) to configure the signaling channels and the Configuration and TimeScale messages are extended from the abstract `Message` class. DCM, UCM and PN messages are integrated into the SAAM packet format as follows.

a) DCM Class

DCM is assigned a message type value of 4. The format for DCM is shown in figure 4.4. In the figure, the second line displays the type of data carried inside a DCM while the first line displays the length of each data field in bytes. Each of these fields is explained below.

- *Type* must be equal to "4" in order to differentiate DCM from other messages.
- *Flow Id* carries the unique flow id assigned to all router-bound signaling traffic for the server that has created the DCM.
- *Server Id* must equal to the router id of the server node.
- *Metric Type* is used to select the metric (transit delay, hop count, etc) that should be used in configuration of the signaling channels.
- *Output Interface* is the IPv6 address of the interface used for forwarding the DCM.
- *Cost To Server* is the cumulative cost(in terms of the selected metric) that DCM has incurred so far starting from the server.

- *Global Timer* is the value used by a recipient of the DCM to set its UCM global timer.
- *Sequence Number* is a value representing the configuration cycle for which DCM is used. Initialized to 0, it is incremented by 1 at the server at start of each refresh interval.

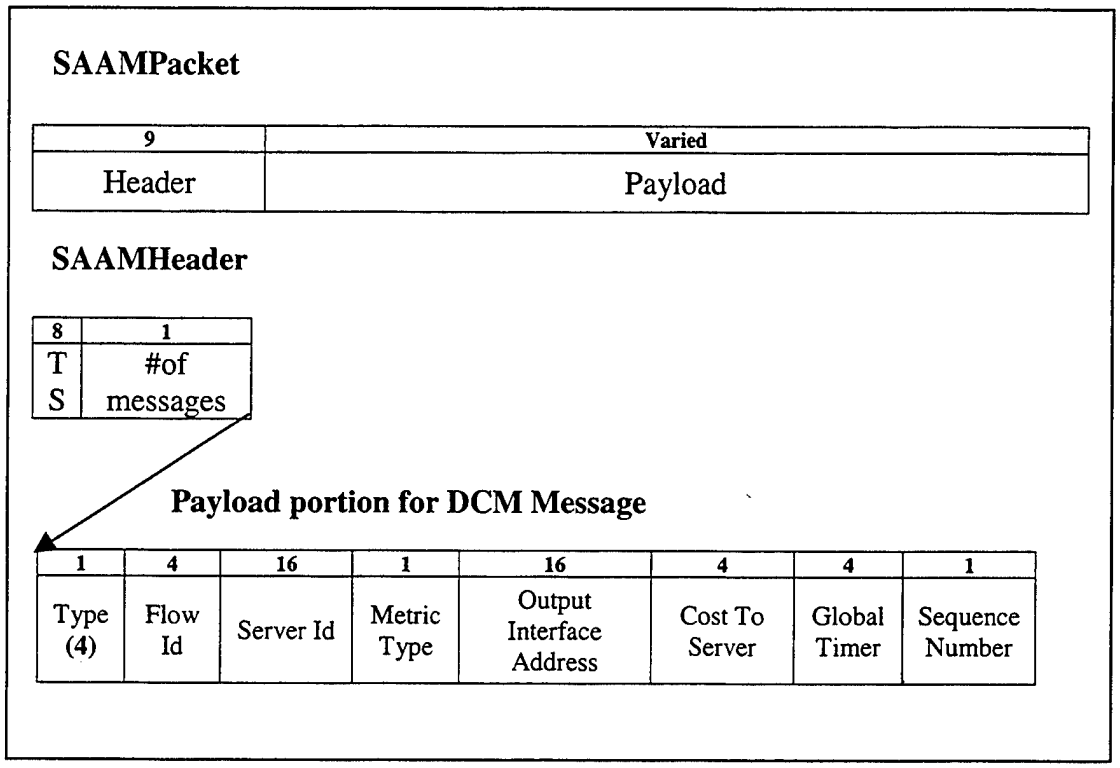


Figure 4.4 DCM Embedded In SAAM Packet.

DCM is used for configuration of server-bound signaling channels. It should be noted that the *Cost To Server* field inside the DCM is not employed by the current version of SCCP. It is placed in the message in order to provide the flexibility for future extensions.

b) UCM Class

The UCM is assigned message type five. A sample UCM is shown in figure 4.5. In the figure, the second line displays types of data carried inside a UCM while the first line displays the length of each information field in bytes. The information carried by the message is used for the following purposes.

SAAMPacket

9	Varied
Header	Payload

SAAMHeader

8	1
T S	#of messages

Payload portion for UCM Message

1	4	16	1	16	1
Type (5)	Flow Id	Output Interface Address	No Of Descendant Routers (N)	Descendant Router Ids	Sequence Number

Figure 4.5 UCM Embedded In SAAM Packet.

- *Type* is the value assigned to UCM to differentiate it from other messages.

- *Flow Id* is the value assigned to server-bound traffic that configures router-bound signaling channels.

- *Output Interface Address* is the IPv6 address of the interface on the router from which UCM has originated.

- *Number of Descendant Routers* is the number of descendant routers to which a router serves as parent.

- *Descendant Router Ids* is IPv6 address of descendant routers that are reachable from the source router of UCM.

- *Sequence Number* is a value representing the configuration cycle in which UCM is used.

c) PN Message Class

The PN message is assigned message type six. The PN message format is shown in figure 4.6. The first line of figure shows the length of each field inside a PN message in bytes while the second line displays the type of data carried inside the message. The information carried by the message is used for the following purposes.

SAAMPacket

9	Varied
Header	Payload

SAAMHeader

8	1
T S	#of Updates

Payload portion for ParentNotification Message

1	4	16	1
Type (6)	Flow id	Source Router Id	Sequence Number

Figure 4.6 PN Message Embedded In SAAM Packet.

- *Type* is the message type value assigned to PN messages to differentiate them from other messages.
- *Flow Id* is the value assigned to traffic that configures router-bound signaling channels.
- *Source Router Id* is the id of the router that sends this PN message.
- *Sequence Number* is the value representing the configuration cycle for which PN message is used.

3. Messages Designed To Add Flexibility To SCCP

The need for *Configuration* and *TimeScale* messages has emerged while integrating SCCP to the existing SAAM emulation software. In order to configure distinct signaling channels for each server, a number of parameters need to be set at each server at initialization of a test-bed. Configuration message is developed to set these parameters from the *DemoStation* instead of hard-coding them in the server code.

On the other hand, a TimeScale message is developed to fine tune the time critical processes (i.e. local and global timers, refresh interval, etc.) to match the CPU power of the hardware platform.

These messages make the emulation software more flexible by permitting easily change of variables rather than changing the related part of the code each time a change is requested. Details of these messages are as follows.

a) Configuration Message Class

The configuration message uses default message type (which is one). The data carried by a Configuration message and length of each data fields in bytes is shown

in figure 4.7. Information carried inside the Configuration message is used for the following purposes.

SAAM Packet

9	Varied
Header	Payload

SAAM Header

8	1
T S	#of Updates

Payload portion for Configuration Message

1	1	4	1	4	4
Type	Server Type	Flow Id	Metric Type	Refresh Interval	Global Time

Figure 4.7 Configuration Message Embedded In SAAM Packet.

- *Type* value for Configuration messages is "1", corresponding to the default message type.

- *Server Type field* identifies whether the server that receives the configuration message will function as primary or backup. Currently, value '0' is assigned to primary server and '1' is assigned to backup server.

- *Flow Id* field represent the value assigned to router-bound signaling traffic originating from the server. In the current implementation of SCCP, typically value '1' is used with primary server and '3' is assigned to backup server.

- *Metric Type* field identifies the algorithm that will be used to configure the signaling channels.

- *Refresh Cycle Time* field represents duration of time interval in milliseconds after which signaling channels will be re–configured periodically.

- *Global Time field* represent the value that will be advertised by the server to the neighbor routers in order to set their global timers.

The metric type field adds flexibility to SCCP. As different algorithms are developed to configure signaling channels, the algorithm to run in SAAM emulation software needs to be determined. The *Configuration* message specifies the algorithm to execute eliminating any need of hard-coding.

b) TimeScale Class

The TimeScale message is assigned the default message type too. It only carries a single data field called TimeScale. This field carries an integer factor (e.g., 100) for scaling network time parameters (e.g., UCM global and local timers) to match the hardware speed of the platform on which emulation software is running. One TimeScale message is sent to each SAAM node when the topology is initialized. The ability to scale time parameters adds great flexibility to test emulation software on a topology made of hosts with very different hardware speeds.

SAAM Packet

9	Varied
Header	Payload

SAAM Header

8	1
T S	#of Updates

Payload Of TimeScale Message

1	4
Type	Time Scale

Figure 4.8 TimeScale Message Embedded In SAAM Packet.

C. CLASSES ADDED TO SAAM EMULATION SOFTWARE FOR IMPLEMENTION OF SCCP

Besides the messages that are already discussed, a number of classes have been added to existing SAAM emulation software for implementation of SCCP (see Appendix D). These classes process the configuration messages in accordance with the SCCP algorithm to establish the signaling channels for each server deployed in the SAAM region.

1. ServerInformation Class

In a SAAM region with multiple servers it is vital not to use parameters associated with one server for another server. For this reason *ServerInformation* class is designed to store the following parameters that need to be kept per server basis at each router.

- *notificatonBuffer* is a *Vector* class object that is used to store the id's of child routers whose PN messages are received.

- *reachableRoutersBuffer* is a Vector class object that is used to store the ids of descendent routers that are learned upon receive of UCMs from child routers.

- *globalTimer* is a Timer class object that is used to prevent infinite wait for a UCM from a child router.

- *localTimer* is a Timer class object too. It is used to determine whether a router is a leaf. A leaf router should not wait for a PN message. Upon expiration of local timer the router will conclude that it is a leaf router.

- *lastSqHeard* field is used to store the last sequence number heard from the server. The sequence number field inside DCM, UCM and PN messages are compared with that stored value to determine whether they are fresh or obsolete old messages. A router deduces that signaling channels are re-configured when it receives a DCM with a higher sequence number value than the stored value.

- *serverDCMReceived* is a boolean variable. It represents whether a DCM is or is not received from the server for the current configuration cycle.

- *serverFlowId* is an integer value. It holds the flow id assigned to the server for which ServerInformation class object is created.

- *numberOfChildren* is an integer value. It holds the number of descendant routers a router has.

The *serverFlowId* data member links a ServerInformation class object to a server. The AutoConfigurationExecutive class accesses and manipulates data members and methods of ServerInformation class while processing DCM, UCM and PN messages. The

flow id field inside these messages is used as the key to determine the appropriate ServerInformation object to manipulate during processing.

2. AutoConfigurationExecutive Class

The majority of functions that are performed by SCCP are implemented in *AutoConfigurationExecutive* class. It receives DCM, UCM and PN messages from *ControlExecutive* class and processes them. The following tasks are executed during processing of these messages.

a) Processing DCM

When a DCM is received, a router first determines whether it knows the server sending it or not. This is accomplished by checking the flow id field inside the DCM. If it is the first DCM heard from the server, the router creates a *ServerInformation* and *RouterBoundCtrlTable* class objects. The *ServerInformation* object is used during process of configuration messages. On the other hand, *RouterBoundCtrlTable* object is used to store the router-bound signaling channel entries.

Upon checking the flow id and creating the aforementioned objects if necessary, the sequence number inside the DCM is checked to decide whether to process or drop it. The message is processed if it carries a higher sequence number than the *lastSqHeard* data member of the *ServerInformation* object. Any DCM with an old sequence number is dropped without further concern.

If a DCM satisfies the conditions to be processed, the router first places an entry into the *FlowRoutingTable* of the server sending the message. That entry is associated with the server-bound signaling channel and it is used to handle server-bound signaling traffic. It is created using the *previous node address* and flow *id* fields inside

the DCM. It should be noted that as a convenience the flow id of the server-bound signaling channel entry is set one higher than the flow id learned from the DCM. Upon completing configuration of server-bound signaling channel, the router sends a PN message to its parent router and DCMs to all its neighbor routers except the parent.

Process of DCM, is completed by setting the local and global timers. A local timer value of 40 milliseconds is hard-coded in the current emulation software. It is adjusted to the platform using the *TimeScale* class object discussed earlier. The value of the local timer is basically proportional to the transmission delay between the child and the parent routers and processing delay at the child router. On the other hand determination of the global timer is more complex since it is related to the distance of the parent from its farthest leaf child rather than neighboring child routers. In current implementation, a maximum global time value of 200 milliseconds is sent to each server inside configuration message. The server advertises this value to its neighbors inside the DCM. From there on each router decreases the global time value by 30 milliseconds and advertises it inside the DCM to the neighbors.

b) Processing PN Message

When a router receives a PN message, first it determines the *ServerInformation* object to manipulate during processing. This is achieved by use of the flow id field inside the message. Then, the router checks the sequence number inside the message to be sure that the message belongs to the current configuration cycle. If the PN message carries a sequence number that is equal to the *lastSqHeard* data member of ServerInformation object, the id of child router inside the message is placed into the *notificationBuffer*. A PN message with an old sequence number is dropped. Moreover,

the router cancels its local timer if it is receiving a PN message for the first time for the current configuration cycle.

c) Processing UCM

Similarly to PN message processing, a router first determines the appropriate *ServerInformation* object to manipulate during process of a UCM message. It then checks the sequence number inside the UCM to make sure it is belong to the current configuration cycle. If the sequence number inside the message is equal to the last sequence number stored for the server for which the UCM is deployed, the id of the child router which sent the UCM is searched for in the *notificationBuffer*. This is done to ensure that UCM is received from a child router.

Upon successfully performing the sequence number and source id checks, the router places an entry into the *RouterBoundControlChannelTable* for all descendant routers it has learned of from the UCM. These entries are the router-bound signaling channels to handle router-bound signaling traffic. After establishing the channels, ids of descendant routers are stored in the *reachabilityBuffer* in order to obtain them easily should the router decides to send a UCM to its parent.

Receipt of UCMs from all children or expiration of the local or global timer triggers sending of a UCM. Therefore, the router receives those descendant router ids from the *reachabilityBuffer* and appends its own id to them. It places these ids into a UCM and sends it to parent router.

3. TimerHandler Class

TimerHandler is an inner class of *AutoControlExecutive* that implements *ActionListener* interface. It triggers UCM sending upon expiration of local or global timer.

4. ServerTableEntry Class

A ServerTableEntry class (see figure 4.9) has *flowId, RouterBoundControlChannelTable* and *serverAddress* data members. A *ServerTableEntry* class object links a server to its RBCCT.

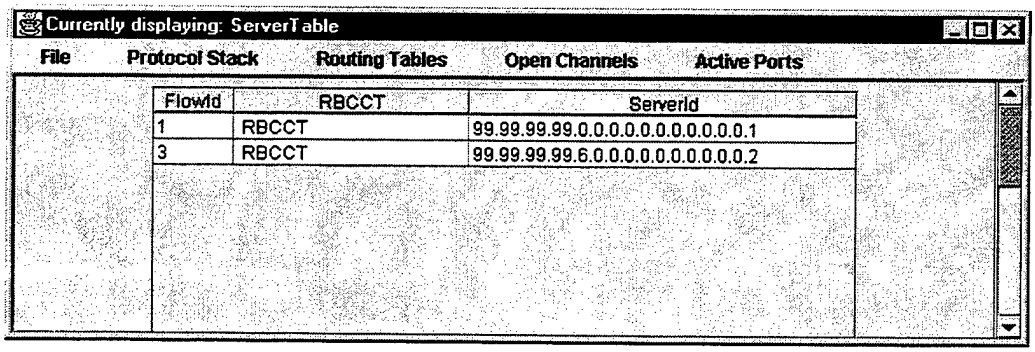


Figure 4.9 Sample ServerTable.

5. ServerTable Class

ServerTable class holds the *ServerTableEntry* objects in a Vector type data structure. It implements the *TableResidentAgent* interface and displays the entries in a GUI (Graphical User Interface) as shown in figure 4.9.

6. RouterBoundCtrlChTableEntry Class

RouterBoundCtrlChTableEntry class object (see figure 4.10) corresponds to a router-bound signaling channel. It has *serverFlowId, destRouterID, nextHop* and *goodness* data members. The *serverFlowId* data member is used as the key to determine which RBCCT a router-bound signaling channel entry should be placed. The *goodness* field is an integer value that identifies the sequence number of the configuration cycle during which the entry is obtained.

7. RouterBoundCtrlChTable Class

RouterBoundCtrlChTable class extends *Hastable* and implements *TableResidentAgent* classes. It holds RouterBoundCtrlChTableEntry class objects in hash table type data structure and displays the stored entries by GUI. A sample RouterBoundCtrlChTable class object with a number of entries is shown in figure 4.10 below.

Currently displaying: Router-bound Control Channel Table (1)

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2	1

Figure 4.10 Sample RouterBoundCtrlChTable.

D. MODIFIED CLASSES

In addition to creation of new classes for deployment of SCCP into the existing SAAM emulation software, a number of new methods have been introduced and a number of existing methods have been modified in the following classes (see Appendix E).

1. ControlExecutive Class

a) ControlExecutive ( )

The method has been modified to instantiate *AutoConfigurationExecutive* class and *ServerInformation* class objects.

b) boolean getIsServer ( )

This method is added to return whether a host in the test-bed is simulating a server or a router. A host by default runs as a router. It is detrmined to be a server when it receives *ServerAgent* class byte codes from *DemoStation* during initialization.

c) IPv6Address getRouterId ( )

SCCP uses *router id* to discriminate routers. It defines router id to be the maximum interface address on the router. Router id is determined during the installation of interfaces from the *DemoStation*. This method is added to return the id of the router.

d) ServerTable getServerTable ( )

This method is added to return the *ServerTable* class object for other classes that require access to it.

e) ***int getTimeScale( )***

Control Executive class receives a Timescale message from the DemoStation and processes it by initializing the integer data member called timeScale. This method is added to return the timeScale data member for the classes that requires to access to it.

f) ***void processMessage (Message message)***

This method processes the messages that ControlExecutive receives from *PacketFactory*. It has been modifies to handle the DCM, UCM, *ParentNotification* and *TimeScale* messages. While the method completely processes the *TimeScale* message it passes *DCM, UCM* and *ParentNotificaion* messages to the *AutoConfigurationExecutive* class.

g) ***void receiveEvent (SaamEvent se)***

This method is modified to set the boolean variable *isServer* to *true* when a packet containing *ServerAgent* class byte codes is received from the DemoStation. Since only servers are sent this packet it has been used as the criteria to distinguish the servers and routers.

h) ***void sendDCM(Object sender, DCM message, int flowID, short sourcePort, IPv6Address destHost, short destPort)***

This method is added for sending a DCM.

i) ***void sendPN(Object sender, ParentNotification message, int flowID, short sourcePort,IPv6Address destHost ,short destPort)***

This method is added for sending a *ParentNotification* message.

j) ***void sendUCM (Object sender, UCM message, int flowID, short sourcePort, IPv6Address destHost, short destPort)***

This method is added for sending a UCM.

k) ***void standUpInterface(InterfaceID id)***

The routers and servers in SAAM region learn the addresses of interfaces on them from the *DemoHello* message they receive from DemoStation during initialization. This method has been modified to determine the id of the router/server. Initially the first received interface address is taken as the id. Later as other interfaces are initialized they are compared to the id already in place. If the new interface has higher IPv6 address it is designated as the id. As a consequence of interface failure id may not be the highest IPv6 address at all times. The current implementation employs the same id and ignores the affect of interface failures on id change.

l) ***IPv6Address getServerBoundNextHop( )***

This method is cancelled from the existing emulation software. It was implemented before deployment of SCCP in order to access the hard coded next hop address for server-bound messages.

2. PacketFactory Class

a) ***void appendDCM(DCM downward)***

This method has been added to put a DCM into byte array format.

b) ***void appendPN(ParentNotification pn)***

This method has been added to put a ParentNotification message into byte array format.

c) ***void appendUCM(UCM ucm)***

This method has been added to put a UCM into byte array format.

d) ***byte [] getDCMBytes( )***

getDCMBytes method has been added to return the byte array that represents a DCM.

e) ***byte [] getPNBytes ( )***

getPNBytes methods has been added to return the byte array that represents a ParentNotification message.

f) ***byte [] getUCMBytes( )***

getUCMBytes method has been added to return the byte array that represents a UCM.

g) ***void processPacket( )***

The method has been modified in order to pass DCM, UCM and ParentNotification messages to ControlExecutive class for further processing.

3. RoutingAlgorithm Class

a) ***void forwardPacket(IPv6Packet packet)***

ForwardPacket method has been modified to conform to the new routing table formats introduced upon deployment of SCCP. In SAAM, traffic requiring QoS and traffic used for signaling purposes carries nonzero flow id while best-effort traffic carries flow id of zero. ForwardPacket method forwards packets based on the flow id field inside them. It first determines which routing table to query to find out the next-hop address. If the flow id of a packet is equal to the flow id assigned to server-bound signaling channels the next-hop address is determined by querying *FlowRoutingTable*. On the other hand, if

the packet carries the flow id that is assigned to router-bound signaling channels, RBCCT of the corresponding server is queried for determination of next hop address.

4. ServerAgentSymetric Class

This class is just renaming of existing ServerAgent class in existing emulation code. It is intended to represent the situation when physical links are assumed to incur the same delay to packets in both directions.

a) void install (ControlExecutive controlExec)

This method is modified to call the *autoconfig( )* method of *Server* class.

b) void processMessage (Message message)

The processMessage method is modified to process the *Configuration* message received by the server during initialization. Upon receipt of a Configuration message, it is passed *processConfiguration( )* method of *Server* class for further processing.

5. Server Class

a) void autoConfig( )

This method is added to create a java *Thread.* That thread starts configuration of signaling channels by periodic DCM sending to the SAAM region. The thread calls *run*( ) method. Run method sleeps initially. Duration of that sleep is required to be long enough to ensure all the routers in the SAAM region are up and ready to receive messages used in conjunction with SCCP. After this sleep time, the server sends a DCM to neighbor routers. It then sleeps again until the next signaling channel refresh cycle. The refresh interval used in the current implementation of SCCP is hard-coded

inside Configuration message. A refresh interval cycle of 2 minutes has been used during test runs of SCCP. This value is larger than optimal for a SAAM environment. The precise determination of refresh interval is beyond the scope of this thesis and a topic for follow-on thesis work.

b) void processConfiguration ( )

Using the information carried inside the *Configuration* message, the server initializes its *type* (i.e. *Primary* or *Backup*), *flow id* to use inside router-bound signaling packets, *metric type* that represents the algorithm to employ in order to establish the signaling channels, *signaling channel refresh interval* and *global timer* to advertise to the directly connected routers.

c) void sendDown (IPv6Address srcInt,IPv6Address des)

This method is implemented to send a DCM message from the server to neighbor routers.

d) setSeguenceNumberForDcmSending ( )

The *setSeguenceNumberForDcmSending* method increments the sequence number by one and place the incremented sequence number inside the DCM. As a result a DCM with a higher sequence number is sent to the neighbor routers periodically.

e) getSeguenceNumberForDcmSending ( )

This method is added to return the sequence number for the current configuration cycle.

V. TEST OF SCCP

SAAM emulation software has been tested on a sample two-server and four-router topology upon deployment of SCCP. In this chapter, the resulting signaling channels are presented using screen captures of the Graphical User Interface (GUI) displays at the end of the test. First, the sample topology is described. Then each phase of SCCP execution is demonstrated.

A. DESCRIPTION OF TEST TOPOLOGY

SCCP is tested on the topology shown in figure 5.1. Table 5.1 displays the IPv6 addresses, IPv4 addresses and MAC addresses of the nodes in the test topology. The IPv4 and MAC addresses are shown in figure 5.1 as well.

Node Name	Router Id	Ipv4Address	Interface	Interface IPv6 Address
Primary	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1	131.120.8.135	0	99.99.99.99.0.0.0.0.0.0.0.0.
Backup	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2	131.120.8.138	13	99.99.99.99.6.0.0.0.0.0.0.0.
Router A	99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	131.120.8.147	1	99.99.99.99.0.0.0.0.0.0.0.0.
			2	99.99.99.99.1.0.0.0.0.0.0.0.
			3	99.99.99.99.2.0.0.0.0.0.0.0.
			4	99.99.99.99.3.0.0.0.0.0.0.0.
Router B	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	131.120.8.139	5	99.99.99.99.1.0.0.0.0.0.0.0.
			6	99.99.99.99.5.0.0.0.0.0.0.0.
Router C	99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	131.120.9.76	7	99.99.99.99.3.0.0.0.0.0.0.0.
			8	99.99.99.99.4.0.0.0.0.0.0.0.
Router D	99.99.99.99.7.0.0.0.0.0.0.0.0.0.0.1	131.120.9.73	9	99.99.99.99.5.0.0.0.0.0.0.0.
			10	99.99.99.99.2.0.0.0.0.0.0.0.
			11	99.99.99.99.4.0.0.0.0.0.0.0.
			12	99.99.99.99.6.0.0.0.0.0.0.0.

Table 5.1 Node Information Of Sample SAAM Topology.

Each node uses the TCP/IP protocol come with the Windows NT or Windows 2000 operating system. In the figure, there is a primary server and a backup server.

Multiple server deployment in the same SAAM region is motivated by fault tolerance (see reference 9) requirements.

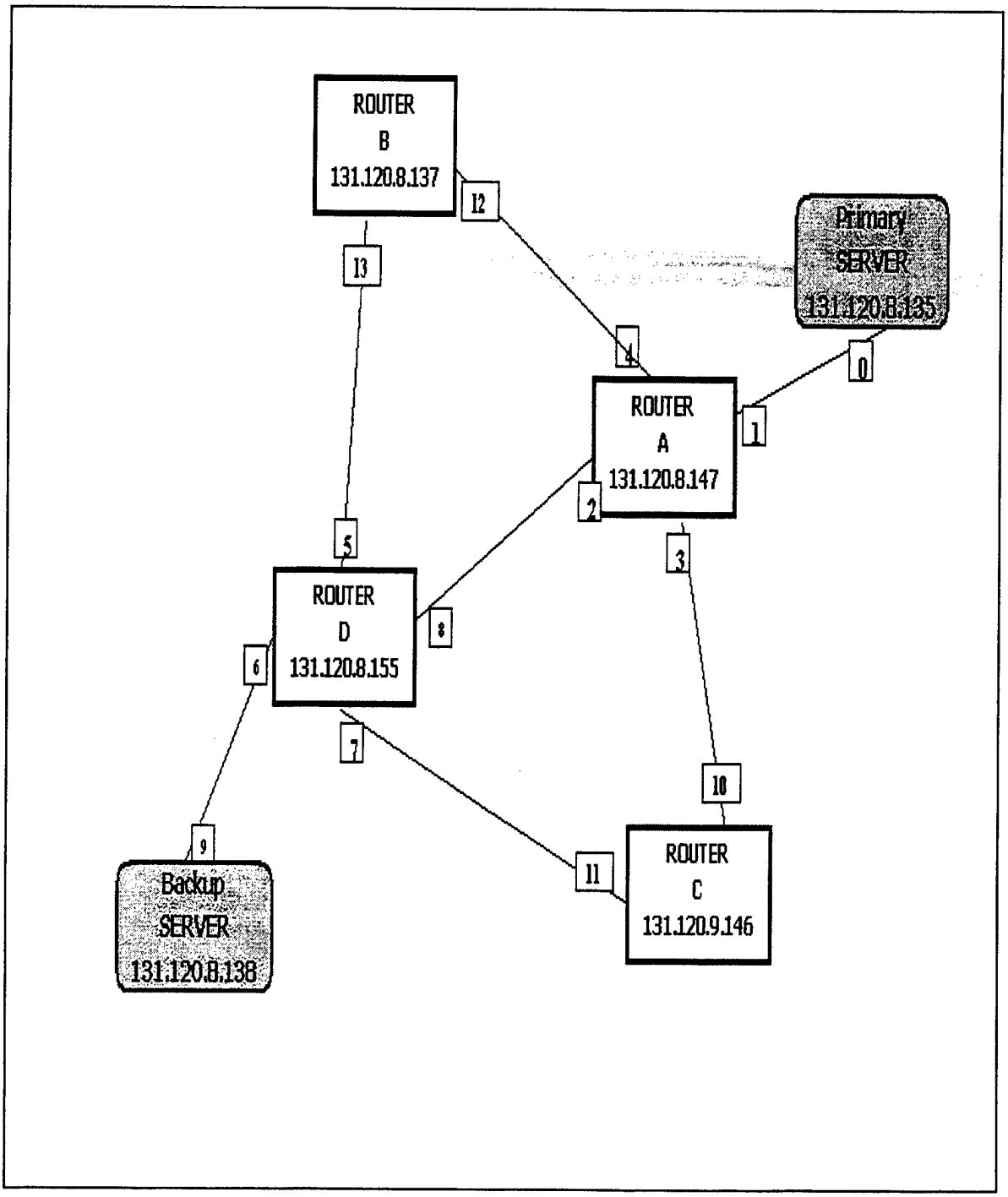


Figure 5.1 Sample Test Topology.

B. TESTING OF SCCP

The purpose of testing is to figure out whether SCCP accomplishes the tasks that it is designed for or not. SCCP is required to perform the following tasks upon execution in order to be regarded successful.

- The router-bound and server-bound signaling channels must be configured for each server in a SAAM region.
- Signaling channels should form a spanning tree rooted at the server. Each router in the SAAM region must be included in the spanning tree.
- Configured signaling channels must not have any loops.
- Configured signaling channels must be consistent. They must take the packets to required destinations rather than misrouting the packets to random destinations. For example, if router B designates router A as parent then router A should have knowledge of router B as its child.
- The periodical re-configuration signaling channels must be performed correctly without mixing channels that have been configured during different refresh cycles.

SCCP has been tested in two phases. The first phase is the initialization of the test-bed. The second phase is the actual running of SCCP that configures the signaling channels. Furthermore, the second phase may be analyzed for a single or multiple refresh intervals. Each of these phases is explained as follows.

1. Initialization of Test-Bed

In order to emulate the SAAM network environment, the nodes need to be initialized with a number of messages and resident agents. A SAAM router is initialized with *InterfaceID, ARPCacheEntry, EmulationTableEntry* and *TimeScale* messages as well as *Scheduler, ARPCache, FlowRoutingTable* resident agents. A SAAM server is initialized with a *ServerAgent* resident agent and a *Configuration* message in addition to the messages and resident agents for a router.

The messages and resident agents are sent to each node from a DemoStation program (see Appendix C) that may run on any host in the physical network. At the end of initialization the routers and servers will have the necessary emulation tables and ARP cache tables to exchange SAAM packets. These tables are described below.

a) Emulation Tables

In SAAM each node interface has an IPv6 address and a MAC address. The physical transport of packets between nodes is accomplished via translation of IPv6 address to IPv4 address by the *Translator* class in the top-level saam package (see reference 7). Each node has an emulation table that stores the mapping between the IPv6 and IPv4 addresses of the neighboring interfaces. Upon processing the *EmulationTableEntry* messages the servers and routersupdate their emulation tables. The results are shown below.

Currently displaying: Emulation Table

File Protocol Stack Routing Tables Open Channels Active Ports

IPv6 NextHop Address	IPv4 Address
99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	131.120.8.147

Figure 5.2 Emulation Table Of Primary Server.

Currently displaying: Emulation Table

File Protocol Stack Routing Tables Open Channels Active Ports

IPv6 NextHop Address	IPv4 Address
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	131.120.8.155

Figure 5.3 Emulation Table Of Backup Server.

Currently displaying: Emulation Table

File Protocol Stack Routing Tables Open Channels Active Ports

IPv6 NextHop Address	IPv4 Address
99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1	131.120.8.135
99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2	131.120.8.137
99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2	131.120.8.155
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2	131.120.9.46

Figure 5.4 Emulation Table Of Router A.

Currently displaying: Emulation Table

File Protocol Stack Routing Tables Open Channels Active Ports

IPv6 NextHop Address	IPv4 Address
99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.1	131.120.8.147
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.2	131.120.8.155

Figure 5.5 Emulation Table Router B.

Currently displaying: Emulation Table

File Protocol Stack Routing Tables Open Channels Active Ports

IPv6 NextHop Address	IPv4 Address
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	131.120.8.147
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.2	131.120.8.155

Figure 5.6 Emulation Table Of Router C.

Currently displaying: Emulation Table

File Protocol Stack Routing Tables Open Channels Active Ports

IPv6 NextHop Address	IPv4 Address
99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.1	131.120.8.147
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	131.120.8.137
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2	131.120.8.143
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	131.120.9.46

Figure 5.7 Emulation Table Of Router D.

b) ARPCache Tables

Each node has an ARPCache table that stores the mapping between the IPv6 and MAC addresses of the neighboring interfaces. Figures 5.8 to 5.13 shows the ARP cache tables which are initialized at the servers and the routers upon processing of the *ARPCacheEntry* messages from the DemoStation.

Currently displaying: ARPCache

File Protocol Stack Routing Tables Open Channels Active Ports

Next Hop	Next MAC
99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	1

Figure 5.8 ARPCache Table Of Primary Server.

Currently displaying: ARPCache

File Protocol Stack Routing Tables Open Channels Active Ports

Next Hop	Next MAC
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	12

Figure 5.9 ARPCache Table Of Backup Server.

Currently displaying: ARPCache

File Protocol Stack Routing Tables Open Channels Active Ports

Next Hop	Next MAC
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2	7
99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2	10
99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2	5
99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1	0

Figure 5.10 ARPCache Table Of Router A.

Currently displaying: ARPCache

File Protocol Stack Routing Tables Open Channels Active Ports

Next Hop	Next MAC
99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.1	2
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.2	9

Figure 5.11 ARPCache Table Of Router B.

Currently displaying: ARPCache

File Protocol Stack Routing Tables Open Channels Active Ports

Next Hop	Next MAC
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	4
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.2	11

Figure 5.12 ARPCache Table Of Router C.

Currently displaying: ARPCache

File Protocol Stack Routing Tables Open Channels Active Ports

Next Hop	Next MAC
99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.1	3
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	8
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2	13
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	6

Figure 5.13 ARPCache Table Of Router D.

2. Execution of SCCP for A Single Configuration Cycle

SCCP configures the signaling channels employing DCM, UCM and PN messages. The overhead introduced to the network by these messages is quite low since these messages are very short (see table 5.2 for the length of each message). While DCM and PN messages have a fixed length, UCM messages are of variable length since the number of descendant routers (denoted by N) varies from router to router.

Message Name	Length (bytes)
DCM	50
UCM	29+(16*N)
PN	25

Table 5.2 Length Of Messages Employed By SCCP.

For the test runs, the local timer duration is hard-coded to 30 milliseconds at each router. The global timer duration of 200 milliseconds is sent to each server from the *DemoStation* via a server *Configuration* message. The server advertises that value to their neighbor routers using DCM. Similarly, each router will reduce the global timer value it receives by 30 milliseconds and advertise the new value to its neighbors using DCM.

At the end of SCCP execution, the server-bound and router-bound signaling channels that are displayed in figure 5.20 to 5.31 have been formed. In addition to routing tables in which signaling channel entries are stored, SCCP employs another table at each node that is called *server table.* The routing table entries for these channels and server tables are shown in the figures that follow.

a) Server Tables

A *Server Table* is used by a node to keep track of the different servers learned via DCM. A server table entry is composed of router-bound flow id of the server, the Router Bound Control Channels Table (RBCCT) associated with that server and address of the server. Using the flow id as key, the node can access and modify the appropriate RBCCT. The following server tables are formed at the routers and the servers.

Currently displaying: ServerTable

File Protocol Stack Routing Tables Open Channels Active Ports

FlowId	RBCCT	ServerId
1	RBCCT	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1
3	RBCCT	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2

Figure 5.14 ServerTable Of Primary Server.

Currently displaying: ServerTable

File Protocol Stack Routing Tables Open Channels Active Ports

FlowId	RBCCT	ServerId
3	RBCCT	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2
1	RBCCT	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1

Figure 5.15 ServerTable Of Backup Server.

Currently displaying: ServerTable

File Protocol Stack Routing Tables Open Channels Active Ports

FlowId	RBCCT	ServerId
1	RBCCT	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1
3	RBCCT	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2

Figure 5.16 ServerTable Of Router A.

Currently displaying: ServerTable

File Protocol Stack Routing Tables Open Channels Active Ports

FlowId	RBCCT	ServerId
1	RBCCT	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1
3	RBCCT	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2

Figure 5.17 ServerTable Of Router B.

Currently displaying: ServerTable

File Protocol Stack Routing Tables Open Channels Active Ports

FlowId	RBCCT	ServerId
1	RBCCT	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1
3	RBCCT	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2

Figure 5.18 ServerTable Of Router C.

Currently displaying: ServerTable

File Protocol Stack Routing Tables Open Channels Active Ports

FlowId	RBCCT	ServerId
1	RBCCT	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1
3	RBCCT	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2

Figure 5.19 ServerTable Of Router D.

b) Server-Bound Signaling Channel Tables

The server-bound signaling channel table entries are established via DCMs and resulting routing table entries are stored in *FlowRoutingTable*. They are shown below in figures 5.20 to 5.25.

Currently displaying: FlowRoutingTable

File Protocol Stack Routing Tables Open Channels Active Ports

Flow ID	SL	Next Hop
4	0	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2

Figure 5.20 Server-Bound Signaling Channel Table Of Primary Server.

Currently displaying: FlowRoutingTable

File Protocol Stack Routing Tables Open Channels Active Ports

Flow ID	SL	Next Hop
2	0	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1

Figure 5.21 Server-Bound Signaling Channel Table Of Backup Server.

Currently displaying: FlowRoutingTable

File Protocol Stack Routing Tables Open Channels Active Ports

Flow ID	SL	Next Hop
4	0	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2
2	0	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1

Figure 5.22 Server-Bound Signaling Channel Table Of Router A.

Currently displaying: FlowRoutingTable

File Protocol Stack Routing Tables Open Channels Active Ports

Flow ID	SL	Next Hop
4	0	99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.2
2	0	99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.1

Figure 5.23 Server-Bound Signaling Channel Table Of Router B.

Currently displaying: FlowRoutingTable

File Protocol Stack Routing Tables Open Channels Active Ports

Flow ID	SL	Next Hop
4	0	99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.2
2	0	99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1

Figure 5.24 Server-Bound Signaling Channel Table Of Router C.

Currently displaying: FlowRoutingTable

File Protocol Stack Routing Tables Open Channels Active Ports

Flow ID	SL	Next Hop
4	0	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2
2	0	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.1

Figure 5.25 Server-Bound Signaling Channel Table Of Router D.

c) *Router Bound Signaling Channel Tables*

Router-bound signaling channel tables are configured via UCM and PN messages. Configured RBCCTs are shown in figures 5.26 through 5.31. The top table of each figure top displays the signaling channels that are configured for the primary server. The table at the bottom displays the signaling channels that are configured for backup server.

Currently displaying: Router-bound Control Channel Table (1)

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	1

Currently displaying: Router-bound Control Channel Table (3)

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness

Figure 5.26 Router-Bound Signaling Channel Tables Of Primary Server.

Currently displaying: Router-bound Control Channel Table (3)

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	1
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	1
99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	1
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	1
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	1

Currently displaying: Router-bound Control Channel Table (1)

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness

Figure 5.27 Router-Bound Signaling Channel Tables Of Backup Server.

Currently displaying: Router-bound Control Channel Table (1)

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2	1
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2	1

Currently displaying: Router-bound Control Channel Table (3)

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness
99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1	1

Figure 5.28 Router-Bound Signaling Channel Tables Of Router A.

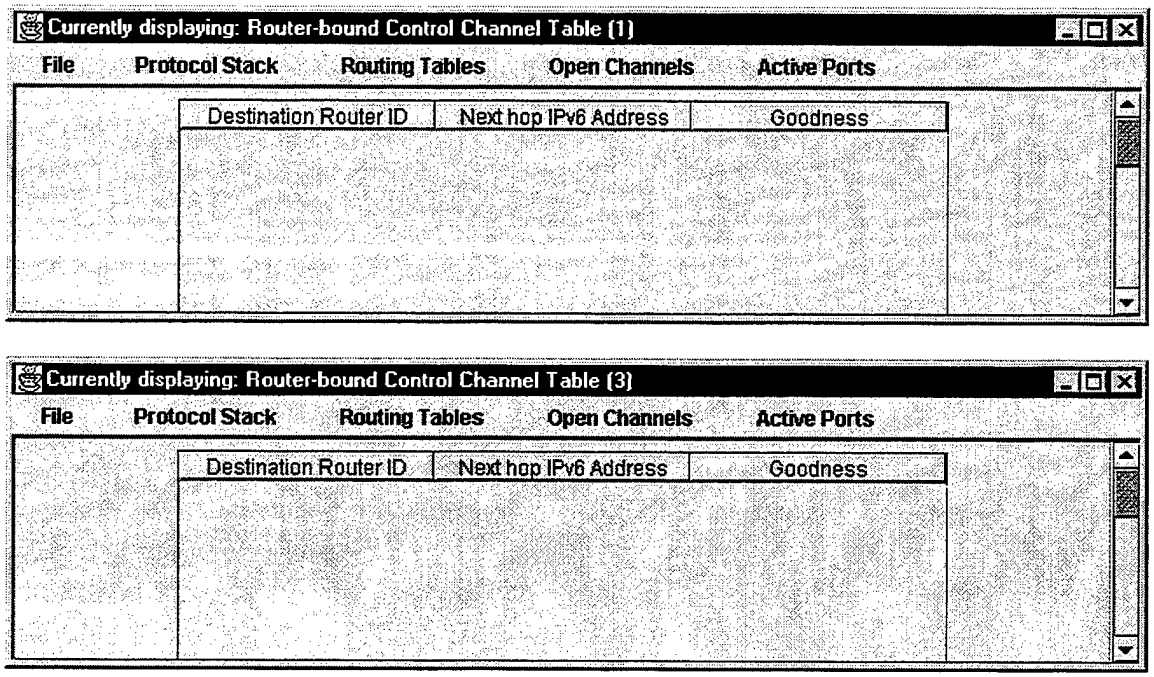


Figure 5.29 Router-Bound Signaling Channel Tables Of Router B.

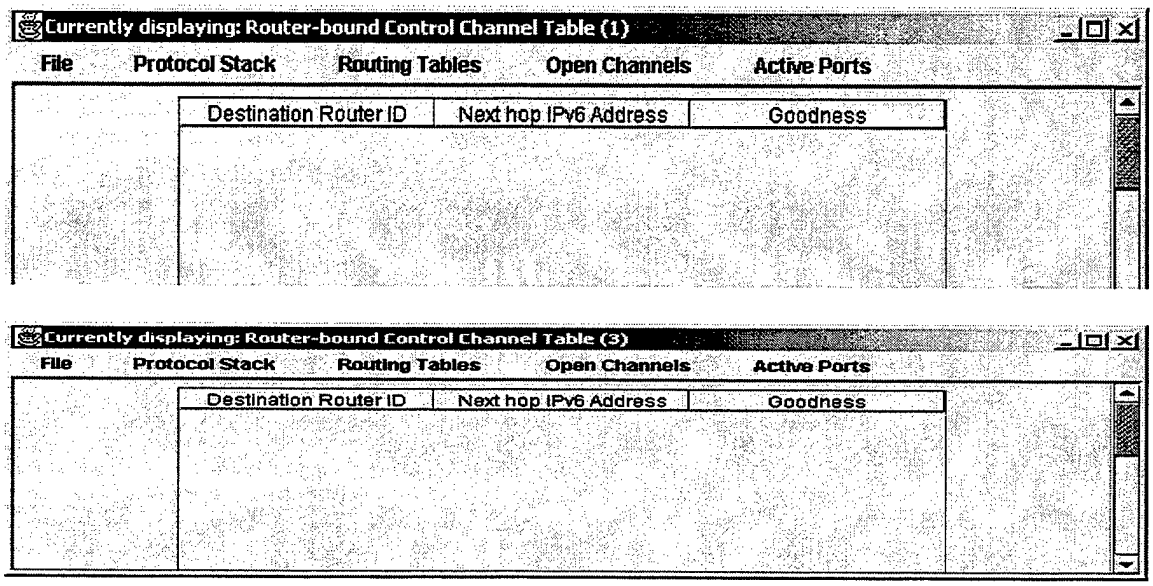


Figure 5.30 Router-Bound Signaling Channel Tables Of Router C.

Currently displaying: Router-bound Control Channel Table (1)

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2	1

Currently displaying: Router-bound Control Channel Table (3)

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.1	1
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	1
99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.1	1
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	1

Figure 5.31 Router-Bound Signaling Channel Tables Of Router D.

A complete and robust configuration of signaling channels has been observed at the end of first SCCP configuration cycle. The servers wait until the start of the next refresh interval to broadcast a DCM with a higher sequence number to re-configure the signaling channels.

3. Configured Signaling Channels

The configured signaling channels upon execution of SCCP at the end of the first cycle are displayed in figure 5.32. The signaling channels that are configured for the primary server are displayed by dashed lines while the signaling channels that are configured for the backup server are displayed by straight lines in the figure.

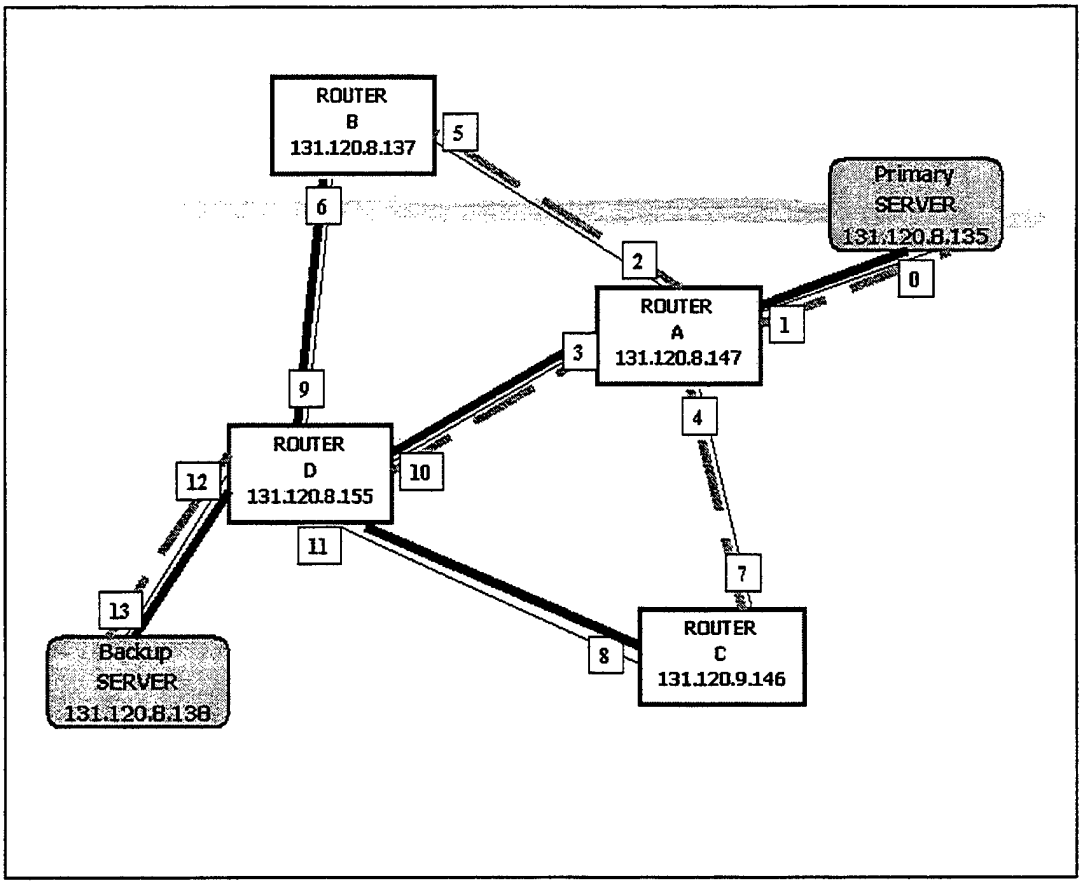


Figure 5.32 Configured Signaling Channels At The End Of First Cycle.

4. Results of Periodical Execution

SCCP has been tested for periodical configuration as well. The periodical refreshment ensures that device and link failures as well as significant load fluctuations will be handled timely. There was no topological change for this test. Therefore the same router and server-bound signaling channels have been configured for all the routers and servers by SCCP at the end of the second configuration cycle. For brevity, only router-bound signaling channels configured for the servers are displayed in figures 5.33 and 5.34 to exemplify the periodic reconfiguration. In the figures, the *goodness* value

represents the current configuration cycle number. It is set to match the sequence number carried by the most recent DCM or UCM message.

1]

:hannels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	2
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	2
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	2
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	2
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2	2

Figure 5.33 Router-Bound Signaling Channel Table Of Primary Server After Configuration Cycle Two.

Currently displaying: Router-bound Control Channel Table [3]

File Protocol Stack Routing Tables Open Channels Active Ports

Destination Router ID	Next hop IPv6 Address	Goodness
99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	2
99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	2
99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	2
99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	2
99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1	99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1	2

Figure 5.34 Router-Bound Signaling Channel Table Of Backup Server After Configuration Cycle Two.

A close look into the tables reveals that SCCP configured signaling channels neatly by forming the expected spanning tree rooted at the servers. All the routers were member of the spanning tree and the signaling channels contained no loop. The parent-child relationship between the routers confirmed that the router and server bound signaling channels were configured consistently. The periodic re–configuration was performed without mixing channels of different refresh cycles. Based of all these results, SCCP is concluded to pass the test successfully.

VI. CONCLUSIONS

A. SYNOPSIS AND CONCLUSION

The main purpose of this thesis is to configure signaling channels within a SAAM region. Signaling channels are required to carry signaling traffic (i.e., traffic carrying control and management information) between the server and the routers. Robust and reliable exchange of signaling messages is vital for successful operation of SAAM.

Available routing protocols that are used in today's networks fall short on timely response to topological changes. Therefore the development of a new routing protocol was inevitable. In order to configure the signaling channels in a SAAM region, SCCP is developed and integrated into the existing SAAM emulation software. SCCP establishes a spanning tree of bi-directional signaling channels rooted at the server. It periodically refreshes the channels to accommodate topological changes in a timely and pro-active manner.

Testing of SCCP support that SCCP meets the specific signaling channel requirements of SAAM. It follows from Chapter 5 that SCCP establishes the signaling channels for each server in a SAAM region in a consistent manner. Additionally, the server-rooted channels add the ability to broadcast or multicast from the server to the routers.

The protocol that has been developed in thesis study has the potential to be used in any multi-service network architecture (e.g., bandwidth broker or ATM PNNI) that employs a node serving a number of other nodes similar to SAAM. Another potential

deployment area for SCCP might be military command and control systems where a base station manages and controls a number of subsidiary stations.

B. LESSONS LEARNED

There are several lessons learned during the thesis study. They originated from the overall nature the SAAM project or specific topic of this thesis.

1. Importance Of Coordination

SAAM is a large-scale project and it covers diverse issues that need to be addressed before it is put into service. Each of these issues was assigned to a team member to investigate.

In the end, the solutions developed by different team members have to be integrated. Therefore extreme attention was required for coordination between team members and control of the overall system structure.

2. Router Id Implementation

To distinguish each node in SAAM, a *router id* definition is developed as part of SCCP. Router id is defined as the highest interface address on the router[4]. Such a definition of router id adds robustness to SCCP. For example, a router may designate a different neighbor router as next hop to server since new channels are built periodically. If a router informs the server with the address of the interface that it uses to forward server-bound signaling traffic, the server and the parent router will interpret the same router as a new router during separate channel configuration cycles. Therefore the router

[4] Router id is called server id when it is used for the server

id that is unique for each router is declared to the parent regardless of which interface is used to forward server-bound signaling traffic.

C. FUTURE WORK

This thesis is only an initial effort for development of a protocol that configures the signaling channels in a SAAM region. The issues that have not been addressed by SCCP need to be solved. Some of the issues that must be worked out are described below

1. Determination of Refresh Interval

SCCP refreshes signaling channels periodically. For testing purposes, SCCP has been run using a refresh interval of 2 minutes. That value was just chosen to demonstrate the execution of SCCP. Different refresh intervals may be required for SAAM regions with different number of nodes or different hardware and software platforms. Therefore precise rules for determining the refresh interval should be developed.

2. Determination of Local and Global Timers

Since SCCP utilizes timers to aggregate messages in order to reduce overhead, duration of these timers need to be determined properly. Currently empirical values have been used for local and global timers. Precise rules for determining these parameters need to be worked out. Such work will increase the robustness and flexibility of SAAM.

3. Implementation Of SCCP With Different Metrics For Configuration of Signaling Channels

Although SCCP has reserved a field inside a DCM to identify the cost metric, the current implementation does not explicitly use any cost metric for determination of the

best route. SCCP regards a link from which the first DCM is received as the best. Furthermore, this approach assumes that a link designated as a channel for server-bound signaling traffic will also perform the best for router-bound signaling traffic in the other direction. Future studies of this topic may focus on eliminating that assumption and using different metrics (e.g., hop count, link bandwidth, etc.) to determine the best path for signaling traffic.

4. Determination of Inter-Region Signaling Channels

Currently SCCP only constructs the signaling channels for the lowest level SAAM region. Therefore enabling SCCP with the capability of determining the signaling channels between different hierarchical levels of SAAM architecture needs to be developed before SAAM can be widely deployed.

APPENDIX A. DEFINITIONS OF NODES IN ROOTED TREES

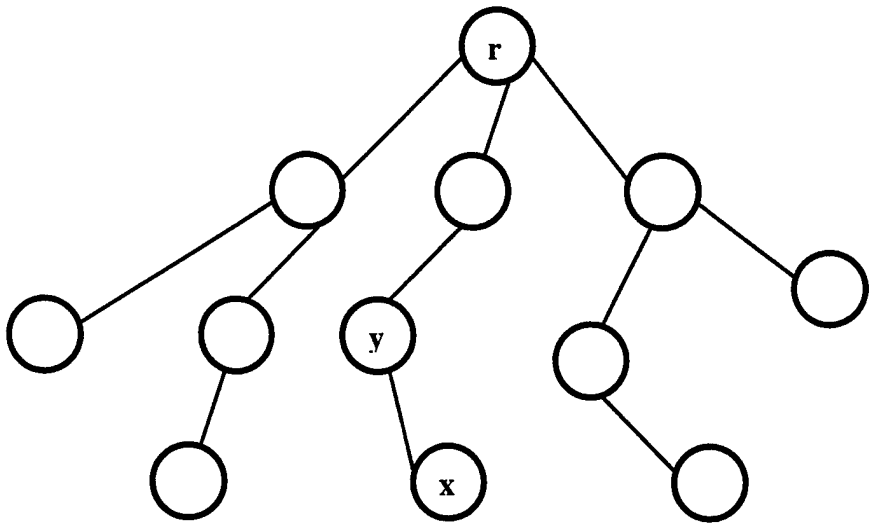


Figure A.1 A Rooted Tree.

A. ROOTED TREES

A ***rooted tree*** is a tree in which one of the nodes is distinguished from the others. The distinguished node is called the ***root*** of the tree.

Consider a node x in a rooted tree T with root r. Any node y in the unique path from the r to x is called an ***ancestor*** of x. If y is an ancestor of x, then x is a ***descendant*** of y.

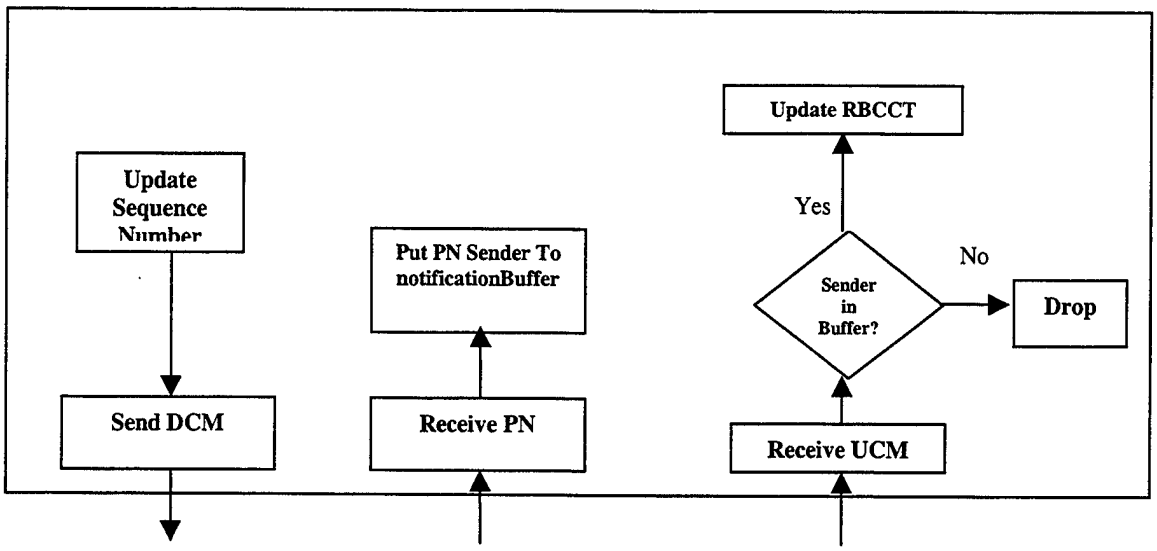
If the last edge on the path from the root r of a tree T to a node x is (y, x), then y is the ***parent*** of x, and x is a ***child*** of y. A node with no children is a ***leaf*** [10].

B. CORRESPONDING DEFINITIONS IN THE SAAM ARCHITECTURE

In SAAM region the server is the root and the signaling channels are rooted at the server.

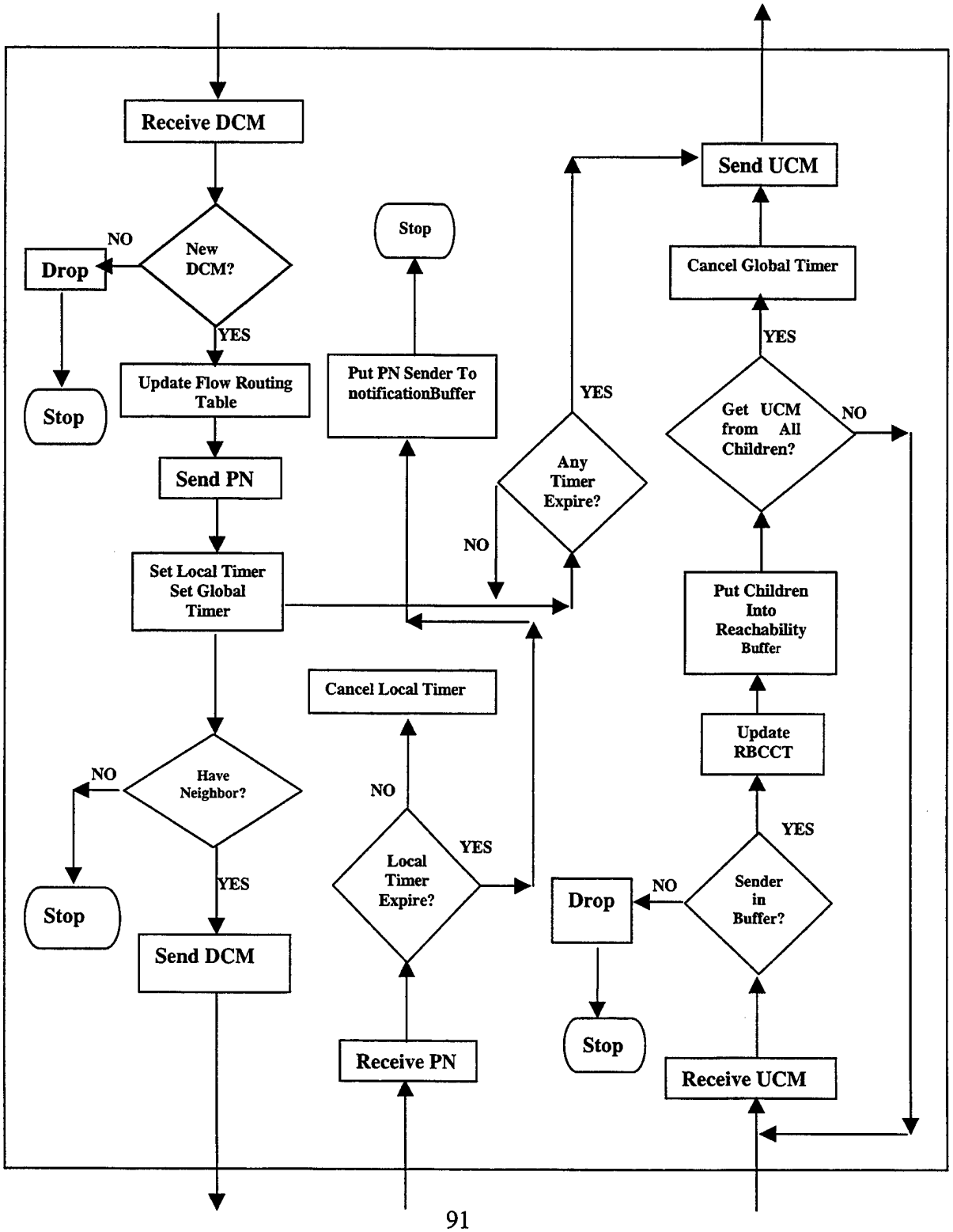
The node that sends the first DCM is the parent while the node that sends the PN message is a child. All routers that are on the same sub-tree rooted at a parent node are descendants while the parent is ancestor of all these nodes. A router that does not receive any PN message is a leaf router.

APPENDIX B. SERVER STATE DIAGRAM



THIS PAGE INTENTIONALY LEFT BLANK

APPENDIX C. ROUTER STATE DIAGRAM



THIS PAGE INTENTIONALY LEFT BLANK

APPENDIX D. CLASSES ADDED TO SAAM FOR SCCP DEPLOYMENT

```
//-----------------------------------------
// Filename : AutoConfigurationExecutive.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project   : SAAM
//-----------------------------------------

package saam.control;

import java.net.UnknownHostException;
import java.net.InetAddress;
import java.util.*;
import java.lang.*;

import saam.router.*;
import saam.net.*;
import saam.message.*;
import saam.residentagent.*;
import saam.residentagent.router.*;
import saam.EmulationTable;
import saam.event.*;
import saam.util.SAAMRouterGui;
import saam.util.Array;

import saam.util.PrimitiveConversions;


import javax.swing.Timer;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class AutoConfigurationExecutive {

/**
* Value simulating router unaware of server initially
*/
private int CTS=10000;

/**
* Values specifying timerhandler clas which timer has expired.
*/
private static final int GLOBAL_IDENTIFIER_FOR_TIMERHANDLER = 1 ;
private static final int LOCAL_IDENTIFIER_FOR_TIMERHANDLER =  0 ;

private Timer localTime,globalTime;
private TimerHandler localHandler, globalHandler;
private ControlExecutive controlExec;
private SAAMRouterGui gui;
private Hashtable ConfigurationTable ;
```

```
/**
* Handles all work relating to autoconfiguration of control channels by
* properly processing the DCM, UCM,PN messages
*/
public AutoConfigurationExecutive(ControlExecutive ce){
    controlExec = ce;
    gui = new SAAMRouterGui(toString());
    ConfigurationTable = new Hashtable(17);

}//AutoConfigurationExecutive()

/**
 * Sets cost of reaching server by learning it from DCM message
 * @param int cts value learned from DCM
 * @return void
 */
 public synchronized void setCurrentCTS (int cts) {
   CTS=cts;
 }

 /**
  * Returns cost of reaching server by learning it from DCM message
  * @return int value
  */

 public synchronized int getCTS(){
   return CTS;
 }

 /**
  * Learning a server from DCM this methods place an entry
  * in the servertable for that server and also creates a
  * RouterBountControlChannel for this server
  * @param int fid flow id of server
  * @param IPv6Address sI id of server
  * @return void
  */
 public synchronized void createNewServerInformation(int fid,
                                                IPv6Address sI){

    RouterBoundCtrlChTable rotBonConTable = new
                   RouterBoundCtrlChTable(fid,1711,controlExec);
    ServerTableEntry entry = new ServerTableEntry
                                        (fid,rotBonConTable,sI);
    controlExec.getServerTable().add(entry);
    ServerInformation serInf = new ServerInformation(fid);
    ConfigurationTable.put( new Integer(fid) ,serInf);
 }// end createNewServerInformation

 /**
  * Process a DCM calling proper method dpending on the metric
  * type field First server table is checked whether the server is
  * known
  * @param DCM dcm DCM message received
  * @return void
```

```
 */
 public synchronized void processDCM(DCM  dcm){
   int fid = dcm.getFlowId();

   if( ! controlExec.getServerTable().hasServer(fid)){
      createNewServerInformation(fid, dcm.getServerId());
   }//end if

   byte metric = dcm.getMetricType();
   if(metric == 0){// Symetric
      processDCMSymmetric(dcm);
   }else{
        //future work for hopcount
   }
}//end processDCM

 /**
  * Process a UCM message, if it has proper seqence number and
  * if sender of UCM has sent PN message. If this is the last
  * UCM waited for, router itself
  * will initiate sending UCM to its parent
  * @param UCM ucm UCM message received
  * @return void
  */
 public synchronized  void processUCM(UCM ucm){

   int fid = ucm.getFlowId();
   int sq = ucm.getSequenceNumber();
   ServerInformation sIn =  (ServerInformation)
           ConfigurationTable.get(new Integer(fid-1));

   if (sq >= sIn.getLastSqHeard()){
     //sending router is last added to the vector or routerids
     IPv6Address senderId = (IPv6Address)
                            ucm.getRouterIds().lastElement();
     boolean contain = false;

     if(sIn.ExistInNotBuffer(senderId)){  //old ta.equals(senderId)
       contain = true;
     }

     if(contain){
       gui.sendText(" Ucm sender is also PN sender  ");
       Vector temp = ucm.getRouterIds();
       Enumeration e = temp.elements();
       while( e.hasMoreElements()){
         IPv6Address rid = (IPv6Address) e.nextElement();
         gui.sendText("Updating recahability with  "+rid.toString() );
         sIn.updateReachableRoutersBuffer(rid);
         //update routerbound control table here
         RouterBoundCtrlChTableEntry ent = new
                   RouterBoundCtrlChTableEntry(fid-1,
                   rid,ucm.getSourceInterfaceAddress(),
                   ucm.getSequenceNumber());
         Message mes = (Message) ent;
```

```
          MessageEvent mEvent = new MessageEvent(this.toString(),
controlExec,controlExec.SAAM_CONTROL_PORT,mes);
          controlExec.receiveEvent(mEvent);

        } // end while
      }//end if contain
    }// if relating SQ number

   if(!controlExec.getIsServer()){
      if( sIn.getNoChildren()== 0 ){
         gui.sendText("Notification Buffer is empty now  ");
         triggerUcmSending(fid-1);
      }else{
         gui.sendText(" Notification IS NOT EMPTY yet ");
      }//end else-if
   }// end of is isServer

   //check for old entries in RBCCT
   ServerTable st = controlExec.getServerTable();
   st.refreshServerTable((fid-1),sq);
 }//end of ProcessUCM

   /**
   * Process a PN message, if it has proper seqence number a
   * Sender o Pn is placed in a vector holding Pn sender for
   * later check for consistence
   * @param PN pn PN message received
   * @return void
   */

  public synchronized  void processPN(ParentNotification pn){

    int fid = pn.getFlowId();
    int sq = pn.getSequenceNumber();
    ServerInformation sIn =  (ServerInformation)
             ConfigurationTable.get(new Integer(fid-1));

    if (sq >= sIn.getLastSqHeard()){

      if(!controlExec.getIsServer()){
        if(sIn.getLocalTime().isRunning()){
          sIn.killLocalTime();
        }
      }// end of server check

      IPv6Address senderId = new IPv6Address();
      senderId = pn.getRouterId();
      sIn.addToNotificationBuffer(senderId);
   }
  }//end of parent notification

   /**
   * Process a DCM message ccarrying symmetric metric type, if it has
   * proper seqence number. Flowroutingtable entry to go
  * server is placed in proper table, PN sent to sender and
```

```
  * DCM is populated from all the interfaces other than received
  * Also timers are set to prevent infinite wait
  * @param DCM dcm DCM message received
  * @return void
  */
  private synchronized  void processDCMSymmetric(DCM dcm){

    int fid = dcm.getFlowId();
    int sq = dcm.getSequenceNumber();
    ServerInformation sIn =
(ServerInformation)ConfigurationTable.get(new Integer(fid));

    if ( !sIn.getServerDCMReceived() & sIn.getLastSqHeard() < sq ){
      // to ensure we start with a clear notification buffer
      sIn.clearNotificationBuffer();
      sIn.setServerDCMReceived(true); // to ignore ones fater the first
      sIn.setLastSqHeard(sq);

      IPv6Address serAd = dcm.getServerId();
      IPv6Address toServer = dcm.getSourceInterfaceAddress();
      byte sl = Interface.CTRL_TRAFFIC_SL;
      FlowRoutingTableEntry entry = new
                   FlowRoutingTableEntry(fid+1,sl,toServer);
      //  Adding  to FlowRoutingTable(entry);
      Message mes = (Message) entry;
      MessageEvent mEvent = new
                    MessageEvent(this.toString(),controlExec,
                    controlExec.SAAM_CONTROL_PORT,mes);
      controlExec.receiveEvent(mEvent);

      ParentNotification pN = new
            ParentNotification(fid+1,controlExec.getRouterId(),sq);
      short sourcePort = (short)controlExec.SAAM_CONTROL_PORT;
      short destPort = (short)controlExec.SAAM_CONTROL_PORT;
      try{
        controlExec.sendPN(controlExec,pN,fid+1,sourcePort,
                                        toServer,destPort);
      }catch(FlowException fe){
        gui.sendText(fe.toString()+" during PN sending ");
      }
      //send DCM From Interfaces

      // For server receiving DCM of other, It does NOT send it to
      //anyone and it does not set a global timer
      if(!controlExec.getIsServer()){
          int gloTimeToSetForThisRouter = dcm.getTimer() -
                            (30*controlExec.getTimeScale());

          populateDCMFromInterfaces(fid,serAd,toServer,
                             dcm.getMetricType(),
                             gloTimeToSetForThisRouter,sq);
        globalHandler =  new TimerHandler(fid,sIn,
                         GLOBAL_IDENTIFIER_FOR_TIMERHANDLER);
        globalTime = new Timer( gloTimeToSetForThisRouter,
                                          globalHandler);
```

```
        sIn.setGlobalTime(globalTime);
        sIn.globalTimeStart();
      }

      localHandler = new
             TimerHandler(fid,sIn,LOCAL_IDENTIFIER_FOR_TIMERHANDLER);
             // zero for local, 1 global
      localTime=new Timer(40*controlExec.getTimeScale(),localHandler );
      sIn.setLocalTime(localTime);
      sIn.localTimeStart();
    }else{
      gui.sendText("A DCM with improper SQ");
    }

  } //end of processDCMSymetric


  private  void processDCMHopCount(DCM dcm){
    //future work
  } //end of processDCMHopcount

  /**
   * Method to send received DM from all the interface on
   * the router except the interface from which it was received
   * @param int fid flowid of server sending this DCM
   * @param IPv6Address serverId Id of server initiating DCM
   * @param IPv6Address tser interace from which the DCM
   * has been received
   * @param byte currentMetric metric used in DCM
   * @param int gloTim global timer set by this router
   * @param int seq current sequence number
   * @return void
   */
  private synchronized void populateDCMFromInterfaces(int sFid,
                    IPv6Address serverId, IPv6Address tSer,
                    byte currentMetric,
                    int gloTim, int seq){
    byte [] networkNotToSend = tSer.getAddress();

    for(int i=0; i< controlExec.getInterfaces().size();i++){
      Interface thisInterface =
                (Interface)controlExec.getInterfaces().get(i);

      int match = 0;
      byte[] outboundInterfaceBytes =
                thisInterface.getID().getIPv6().getAddress();
      int bytesToCheck = 5;

      for(int index=0;index<bytesToCheck;index++){
        if((networkNotToSend[index]&0xFF)==
                (outboundInterfaceBytes[index]&0xFF)){
          match++;
        }//if
      }//for
```

```
      // sending from unmatching interfaces
      IPv6Address destv6Address = null;
      if(match!=bytesToCheck){
        EmulationTable eTable = controlExec.getEmulationTable();
        EmulationTableEntry entry = eTable.lookByNetworkAddress(
thisInterface.getID().getIPv6());
        destv6Address = entry.getNextHopIPv6();
        IPv6Address sI = thisInterface.getID().getIPv6();
        DCM dcmToDown = new DCM(sFid,serverId,currentMetric,sI,
                                this.getCTS(),
gloTim,seq);
        short srPort = (short)controlExec.SAAM_CONTROL_PORT;
        short dPort = (short)controlExec.SAAM_CONTROL_PORT;
        try{
          gui.sendText("Forwarding DCM to  " +
                 destv6Address.toString() + "on  "+

thisInterface.getID().toStringTEMP());
          controlExec.sendDCM(controlExec,dcmToDown,sFid,srPort,
                                              destv6Address,dPort);
        }catch (Exception fec ){
          gui.sendText(fec.toString());
        }
      }//last if
    }//for

  }//end of populate

   /**
   * Method initiates sending UCM message to the parent of the roueter
   * @param int fidOfServer flowid of server
   * @return void
   */

  private synchronized  void triggerUcmSending(int fidOfServer){

    ServerInformation sIn =  (ServerInformation)
           ConfigurationTable.get(new Integer(fidOfServer));
    //since did not set globalTime for server, need to check whether
    //this is null or not
    if(sIn.getGlobalTime() != null){
      if(sIn.getGlobalTime().isRunning()){
        sIn.killGlobalTime();
      }
   } // end of if testing for null
   try{
     Message message = (Message)(new
                       FlowRoutingTableEntry(fidOfServer+1));
     IPv6Address upToServer=
                 controlExec.getRoutingAlgorithm().
                 getFromFlowRoutingTable(message).getNextHop();
     IPv6Address sInter = controlExec.getRoutingAlgorithm().
                              lookByNetworkAddress(upToServer);
     //finally as last element add the own router id
     sIn.updateReachableRoutersBuffer(controlExec.getRouterId());
```

```
      UCM toSendUp = new
         UCM(fidOfServer+1,sInter,
                sIn.getReachableRoutersBuffer().size()
                ,sIn.getReachableRoutersBuffer(),sIn.getLastSqHeard() );
      try{
        controlExec.sendUCM( controlExec,toSendUp,
fidOfServer+1, (short)controlExec.SAAM_CONTROL_PORT,
        upToServer,(short)controlExec.SAAM_CONTROL_PORT);
      }catch (FlowException fec ){
        gui.sendText(fec.toString());
      }
      sIn.setServerDCMReceived(false);
    }catch (Exception e ){
      gui.sendText(" PROBLEM "+e.toString());
    }
    // after sending ucm clean buffer holding reachable routers
    sIn.getReachableRoutersBuffer().clear();

  }// end of trigger UCM sending

   /**
    * Returns a <code>String</code> representation of this object
    * @return The <code>String</code> representation of this object
    */
   public String toString(){
     return ("AutoConfigurationExecutive");
   } // end of toString()

   /**
    * Inner class to initiate UCM sending in case a timer expires
    */
   class TimerHandler implements ActionListener{
      int data;
      ServerInformation sInfo;
      int localORglobal; // to determine local or global

      public TimerHandler(int fd, ServerInformation so,int lg){
        data=fd;
        sInfo = so;
        localORglobal = lg;
      }

      public void actionPerformed(ActionEvent event){

        if (localORglobal == 0){
          sInfo.killLocalTime();
        }else if( localORglobal == 1){
          sInfo.killGlobalTime();
        }
        triggerUcmSending(data);
         }

   }// end of TimerHandler
 } //end of class AutoControlExecutive
```

```
//-----------------------------------------
// Filename : ServerInformation.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project   : SAAM
//-----------------------------------------

package saam.control;

import java.util.Vector;
import java.util.Enumeration;
import javax.swing.Timer;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

import saam.util.Array;
import saam.net.IPv6Address;
import saam.util.PrimitiveConversions;


public class ServerInformation {

private Vector notificationBuffer;
private Vector reachableRoutersBuffer;
private Timer globalTime = null;
private Timer localTime= null;
private int sequenceNumber=0;
private int lastSqHeard;
private boolean serverDCMReceived = false;
private int serverFlowId;
private short numberOfChildren = 0;

/**
 *Class to keep specific values for a server. Object of this class
 *is  created for each server. These values are necessary to
 *properly process DCM, UCM,PN messages
 */

 public ServerInformation (int fid){
     notificationBuffer = new Vector();
     reachableRoutersBuffer = new Vector();
     serverFlowId = fid;
 }

 /**
  * Returns id of server
  * @return int value
  */
 public int getServerFlowId(){
    return this.serverFlowId;
 }

 /**
   * Clear the Notification buffer which hold ids of pn senders
   * @return void
```

```
 */
public synchronized void clearNotificationBuffer(){
   notificationBuffer.clear();
}

/**
 * Check whether a an IPv6Address representing the router id is in
 * notification buffer
 * @return boolean value
 */
public synchronized boolean ExistInNotBuffer(IPv6Address adr){
   Enumeration ec = notificationBuffer.elements();

   while(ec.hasMoreElements()){
     IPv6Address  ta = (IPv6Address) ec.nextElement();
     if(ta.equals(adr)){

       if( numberOfChildren > 0){
         numberOfChildren --;
       }
       return  true;
     }
   }// end while
    return false;
}

/**
 * Retruns the number of Pn senders
 * @return int value
 */
public synchronized int getNoChildren(){
  return this.numberOfChildren;
}

/**
  * Returns Notification buffer which hold ids of pn senders
  * @return Vector of router ids
  */
 public synchronized Vector getNotificationBuffer(){
   return notificationBuffer;
 }

 /**
  *Adds an IPv6Address representing a router id to Notification
  * buffer
  * @return void
  */
 public synchronized void addToNotificationBuffer(IPv6Address rd){
   notificationBuffer.add(rd);
   numberOfChildren ++;
 }

 /**
  * Checks whetrher Notification buffer is empty
  * @return void
```

```
   */
  public synchronized boolean NotificationBufferIsEmpty(){
     if(this.notificationBuffer.isEmpty()){
       return true;
     }else{
       return false;
     }
  }

  /**
   * Returns the vector containg the routers reachable by this router
   * @return vector holding router ids.
   */

  public synchronized Vector getReachableRoutersBuffer(){
    return reachableRoutersBuffer;
  }

 /**
  * Add a router id reachablerouters buffer
  * @return void.
  */
public synchronized void updateReachableRoutersBuffer(IPv6Address rid){
    reachableRoutersBuffer.add(rid);
}

/**
 * Returns the last sequnce number heard from the server
 * @return int sequence number value.
 */
 public synchronized int getLastSqHeard(){
     return lastSqHeard;
 }
 /**
  * Sets the sequnce number heard from the server
  * @return void.
  */
  public synchronized void setLastSqHeard(int sn){
    lastSqHeard=sn;
  }

  /**
   * Sets the localtime value for that server
   * @return void.
   */
  public synchronized void setLocalTime (Timer tm){
    localTime = tm;
  }

  /**
   * Stops the globalTimer
   * @return void.
   */
  public synchronized void killGlobalTime (){
     if( globalTime != null){
```

```
      if (globalTime.isRunning()){
        globalTime.stop();
      }
    }
}

/**
 * Stops the localTimer
 * @return void.
 */
public synchronized void killLocalTime (){
    if( localTime != null){
     if (localTime.isRunning()){
        localTime.stop();
     }
    }
}

/**
 * Starts the globalTimer
 * @return void.
 */
public synchronized void globalTimeStart(){
   globalTime.start();
}

/**
 * Starts the localTimer
 * @return void.
 */
public synchronized void localTimeStart(){
   localTime.start();
}

/**
 * Sets the globalTimer
 * @return void.
 */
public synchronized void setGlobalTime (Timer tm){
  globalTime = tm;
}

/**
 * Returns the localTimer
 * @return Timer value.
 */
public synchronized Timer getLocalTime (){
  return  localTime;
}

/**
 * Returns the globalTimer
 * @return Timer value.
 */
public synchronized Timer getGlobalTime (){
```

```
    return globalTime;
  }
  /**
   *Check whether DCM has been heard from the server for the current
   * cycle
   * @return boolean value.
   */
  public synchronized boolean getServerDCMReceived(){
    return  serverDCMReceived;
  }
  /**
   * Sets varibale specifying the DCM received from the server
   * for the current cycle
   * @return void.
   */
  public synchronized void setServerDCMReceived(boolean b){
    serverDCMReceived = b;
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return("ServerInformation");
  }// end toString()

}// end of ServerInformation
```

```
//----------------------------------------
// Filename : DCM.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project   : SAAM
//----------------------------------------

//metricType 0->"First arriving Best Aproach", 1-> Hop Count

package saam.message;

import saam.control.ControlExecutive;
import saam.util.*;
import saam.net.*;
import java.io.*;
import java.net.UnknownHostException;

public class DCM extends Message{


private byte[] bytes;
private static int flowId;
private IPv6Address serverId;
private byte metricType;
private IPv6Address sourceInterfaceAddress;
private int costToServer ;
private int Timer;
private static int sequenceNumber;

/**
 * Constructs a Configuation message with the values provided.
 * @param int rbcFid flowid used by the server sending this DCM .
 * @param IPv6Address svrid id of server sending DCM.
 * @param byte mT metric type to use.
 * @param IPv6Address srcIFaceAddress intercase that DCM leaves.
 * @param int CTS metric value stating cost to server
 * @param int Tmr to set globaltimer at router.
 * @param int Snumber sequence number of the refreshment cycle.
 */

public DCM( int rbcFid,IPv6Address svrid, byte mT,
            IPv6Address srcIFaceAddress,
            int CTS, int Tmr,int sNumber){
    super(Message.DCM_TYPE);
    flowId = rbcFid;
    serverId=svrid;
    metricType = mT;
    sourceInterfaceAddress = srcIFaceAddress;
    costToServer = CTS;
    Timer = Tmr;
    sequenceNumber = sNumber;

    bytes =  Array.concat(type,PrimitiveConversions.getBytes(flowId));
    bytes = Array.concat(bytes,serverId.getAddress());
    bytes = Array.concat(bytes,metricType);
```

```
    bytes = Array.concat(bytes,sourceInterfaceAddress.getAddress());
    bytes = Array.concat(bytes,
            PrimitiveConversions.getBytes(costToServer));
    bytes = Array.concat(bytes,PrimitiveConversions.getBytes(Timer));
    bytes = Array.concat(bytes,
            PrimitiveConversions.getBytes(sequenceNumber));

  }

 /**
   * Constructs a   DCM message with the byte array provided.
   * @param byte [] ent byte array repreesenting DCM message.
   */

  public DCM(byte[] bytes) {

    super(Message.DCM_TYPE);
    this.bytes = bytes;
    flowId = PrimitiveConversions.getInt(
                        Array.getSubArray(bytes,1,5));

    try{
      serverId = new IPv6Address(Array.getSubArray(bytes,5,21));
    }catch(UnknownHostException uhe){
      System.out.println(uhe.toString());
    }
    metricType =this.bytes[21];
    try {
      sourceInterfaceAddress = new
                    IPv6Address(Array.getSubArray(bytes,22,38));
    }catch(UnknownHostException uhe){
       System.out.println(uhe.toString());
    }
    costToServer =
          PrimitiveConversions.getInt(Array.getSubArray(bytes,38,42));
    Timer = PrimitiveConversions.getInt(Array.getSubArray(bytes,
                                                          42, 46));
    sequenceNumber =
          PrimitiveConversions.getInt(Array.getSubArray(bytes,46,50));
  }

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
   */
  public byte[] getBytes(){
    return bytes;
  }
  /**
   * Returns the flowid of server.
   * @return int value
   */
  public int getFlowId(){
    return flowId;
  }
```

```
/**
 * Returns the length of this Message.
 * @return The length of this Message.
 */
public short length(){
  try{
    return (short)bytes.length;
  }catch(NullPointerException npe){
    return 0;
  }
}

/**
 * Returns the cost to reach server .
 * @return int value
 */

public int getCTS(){
  return  costToServer ;
}
/**
 * returns the cycle time value
 * @param int value
 */

 public byte getMetricType(){
   return  metricType;
}

/**
 * returns the global time value
 * @param int value
 */
public int getTimer(){
  return  Timer ;
}

/**
 * returns the sequence number
 * @param int value
 */

public int getSequenceNumber(){
  return  sequenceNumber ;
}

/**
 * returns address of interface from which DCM hhas left the router
 * @param IPv6Address value
 */

public IPv6Address getSourceInterfaceAddress(){
  return  sourceInterfaceAddress ;
}
```

```
  /**
   * returns address of server initiating this DCM
   * @param IPv6Address value
   */
  public IPv6Address getServerId(){
    return  serverId;
  }
  /**
   * Sets address of interface from which DCM hhas left the router
   * @param void
   */
   public void setSourceInterfaceAddress( IPv6Address adr){
     sourceInterfaceAddress = adr;
  }


  /**
   * sets the the cost to server for dcm
   * @return void.
   */
  public void setCTS (int cts) {
    costToServer= cts;
  }

  /**
   * sets the the Timer of dcm
   * @return void.
   */
  public void setTimer (int tm) {
    Timer = tm;
  }


  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
      return "DCM Message ";

  }//end toString()

}//end of DCM.java
```

```
//----------------------------------------
// Filename : UCM.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project   : SAAM
//----------------------------------------

package saam.message;

import saam.util.*;
import saam.net.*;
import java.io.*;
import java.util.*;
import java.net.UnknownHostException;

public class UCM extends Message{


private byte[] bytes;
private int flowId;
private IPv6Address sourceInterfaceAddress;
private int noRouters ;
private Vector routerIDs= new Vector();
private int sequenceNumber;

/**
 * Constructs a UCM message with the values provided.
 * @param int fid flowid used to go server direction.
 * @param IPv6Address srcIFaceAddress address UCM leaves the router.
 * @param int noRtrs no of routers reachable downward.
 * @param Vectot rtrIds ids of routers reachable.
 * @param int Snumber sequence number of the refreshment cycle.
 */
  public UCM(int fid,IPv6Address srcIFaceAddress, int noRtrs,
                                    Vector rtrIDs, int sNumber){
    super(Message.UCM_TYPE);
    flowId = fid;
    sourceInterfaceAddress= srcIFaceAddress;
    noRouters  = noRtrs;
    routerIDs = rtrIDs;
    sequenceNumber = sNumber;

    bytes =  Array.concat(type,PrimitiveConversions.getBytes(flowId));
    bytes =  Array.concat(bytes,sourceInterfaceAddress.getAddress());
    bytes =  Array.concat(bytes,
             PrimitiveConversions.getBytes(noRouters));

    if(!rtrIDs.isEmpty()){
      byte routerIDs [] = new byte[16*noRouters] ;
      byte [] temp;
      for( int i= 0; i<noRtrs; i++){
        IPv6Address ad = (IPv6Address) rtrIDs.elementAt(i);
        temp = ad.getAddress();
        //for first element
```

```
      if(i== 0 ){
        routerIDs = temp;
      }else{
        routerIDs = Array.concat ( routerIDs,temp);
      } //end else
    }//end for

    bytes = Array.concat(bytes,routerIDs);
  } //end if
  bytes = Array.concat(bytes,
          PrimitiveConversions.getBytes(sequenceNumber));

}

/**
 * Constructs a   UCM message with the byte array provided.
 * @param byte [] bytes byte array repreesenting UCM message.
 */
public UCM ( byte[] bytes) {
  super(Message.UCM_TYPE);
  this.bytes = bytes;
  flowId = PrimitiveConversions.getInt
           (Array.getSubArray(bytes,1,5));
  try {
    sourceInterfaceAddress = new
                IPv6Address(Array.getSubArray(bytes,5, 21));
  }catch(UnknownHostException uhe) {
     System.out.println("problem getting sourceInterface Address ");
  }
  noRouters =
        PrimitiveConversions.getInt(Array.getSubArray(bytes,21,25));

  int startindex =25;
  for ( int i = 0 ; i<noRouters ; i++){
    try{
      IPv6Address ta = new
                          IPv6Address((Array.getSubArray(bytes,
                          startindex,startindex+16)));
      routerIDs.addElement(ta);
    }catch(UnknownHostException e){
        System.out.println("ADDRESS PROBLEM IN UCM ");
    }
    startindex = startindex+16;
  } // end of for

  sequenceNumber =
              PrimitiveConversions.getInt(Array.getSubArray(bytes,
              startindex,startindex+4));
}


/**
 * Returns The byte array representation of this Message.
 * @return The byte array representation of this Message.
 */
```

```
public byte[] getBytes(){
  return bytes;
}

/**
 * Returns the number of routers reachable by that router.
 * @return int value .
 */
public int getNoRouters(){
  return noRouters;
}

/**
 * Returns vector that holds ids of routers  reachable.
 * @return Vector .
 */
public Vector getRouterIds(){
  return routerIDs;
}

/**
 * Returns sequence number used in the UCM message.
 * @return Vector .
 */
public int getSequenceNumber(){
  return sequenceNumber;
}
/**
 * Returns IPv6Address that this message letf the router that
 * it is originating from.
 * @return IPv6Address .
 */
public IPv6Address getSourceInterfaceAddress(){
  return sourceInterfaceAddress;
}
/**
 * Returns flowid used in UCM message.
 * @return int value .
 */
public int getFlowId(){
   return flowId;
}


/**
 * Returns the length of this Message.
 * @return The length of this Message.
 */
public short length(){
  try{
    return (short)bytes.length;
  }catch(NullPointerException npe){
    return 0;
  }
}
```

```
  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return "UCM Message ";
  }//end toString()


}
```

```
//----------------------------------------
// Filename : ParentNotification.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project   : SAAM
//----------------------------------------

package saam.message;

import saam.util.*;
import saam.net.*;

import java.util.*;
import java.net.*;
 /**
   * Message use dto inform the DCM sdender that it is chosen as parent
   */

public class ParentNotification extends Message{

private byte[] bytes;
private int flowId;
private IPv6Address sourceRouterId;
private int sequenceNumber;

/**
 * Constructs a PN message with the values provided.
 * @param int fid flowid to go parent node.
 * @param int Snumber sequence number of the refreshment cycle.
 */
 public ParentNotification (int fid, IPv6Address srcRtrId,
                                                  int sNumber){
    super(Message.PARENT_NOTIFICATION_TYPE);
    flowId = fid;
    sourceRouterId= srcRtrId;
    sequenceNumber = sNumber;

    bytes = Array.concat(this.getType(),
             PrimitiveConversions.getBytes(flowId));
    bytes = Array.concat(bytes,sourceRouterId.getAddress());
    bytes = Array.concat(bytes,
            PrimitiveConversions.getBytes(sequenceNumber));
  }

/**
 * Constructs a   PN message with the byte array provided.
 * @param byte [] bytest byte array repreesenting PN message.
 */
 public ParentNotification(byte[] bytes) {
     super(Message.PARENT_NOTIFICATION_TYPE);
     this.bytes = bytes;
     flowId = PrimitiveConversions.getInt(
              Array.getSubArray(bytes,1,5));
     try{
```

```
      sourceRouterId = new IPv6Address(Array.getSubArray(bytes,
                                                          5, 21));
    }catch(UnknownHostException uhe){
      System.out.println(uhe.toString());
    }
    sequenceNumber =
          PrimitiveConversions.getInt(Array.getSubArray(bytes,21,25));

}//end of cons


 /**
  * Returns The byte array representation of this Message.
  * @return The byte array representation of this Message.
  */

 public byte[] getBytes(){
   return bytes;
 }


 /**
  * Returns The router id sending PN.
  * @return Ipv6Addres value .
  */
  public IPv6Address getRouterId(){
   return sourceRouterId;
 }

  /**
  * Returns the flowid of used in Pn message.
  * @return int value
  */
 public int getFlowId(){
   return flowId;
 }

 /**
  * returns the sequence number
  * @param int value
  */
 public int getSequenceNumber(){
   return sequenceNumber;
 }


 /**
  * Returns the length of this Message.
  * @return The length of this Message.
  */
 public short length(){
   try{
     return (short)bytes.length;
   }catch(NullPointerException npe){
     return 0;
```

```
    }
  }


  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
     return "Parent notification Message ";
  }//end toString()

}
```

```
//----------------------------------------
// Filename : RouterBoundCtrlChTableEntry.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project  : SAAM
//----------------------------------------

package saam.message;

import saam.net.IPv6Address;
import saam.util.*;
import java.net.UnknownHostException;

/**
 * A RouterBoundCtrlChTableEntry object contains the destination router
 * ID (IPv6 address), and an IPv6 address representing the next hop.<p>
 * The entry could be either an add or a remove depending on which
 * constructor is used.  An add is required to have all fields, whereas
 * a remove only requires the field that is used as the lookup key in
 * the associated table.
 * Entries can be retrieved as a byte array using the getBytes()
 * method.
 */
public class RouterBoundCtrlChTableEntry extends Message{

private int serverFlowId;
private IPv6Address destRouterID;
private IPv6Address nextHop;
private int Goodness;

private byte[] entry;

/**
 * Constructs an entry for the <em>Router-bound Control
 * Channel Table</em>.
 * @param destRouterID The IPv6 address to be stored.
 * @param nextHop IPv6Address to reach router with corresponding id.
 * @param good DCM cycle value for which the entry is valid.
 */
 public RouterBoundCtrlChTableEntry(int sFId,
                                    IPv6Address destRouterID,
                                    IPv6Address nextHop,int good) {
      serverFlowId = sFId;
      this.destRouterID = destRouterID;
      this.nextHop = nextHop;
      Goodness = good;

      entry =
             Array.concat(PrimitiveConversions.getBytes(serverFlowId),
             destRouterID.getAddress());
      entry =  Array.concat(entry,  nextHop.getAddress());
      entry =  Array.concat(entry,
               PrimitiveConversions.getBytes(Goodness));
    }
```

```
  /**
   * Returns the next hop as an IPv6Address object.
   * @return The next hop as an IPv6Address object.
   */
  public IPv6Address getNextHop(){
    return nextHop;
  }

  /**
   * Returns the destination router ID associated with this Message
   * @return The destination router ID associated with this Message
   */
  public IPv6Address getDestRouterID(){
    return destRouterID;
  }
/**
  * Returns the goodness of the entry
  * @return int value
  */
public synchronized int getGoodness(){
  return Goodness;
}
/**
  * Returns the serverflowid for which that RBCCT belongs to
  * @return int value
  */
 public int getServerFlowId(){
   return serverFlowId;
 }

  /**
   * Returns The RouterBoundCtrlChTableEntry as a byte array with
   *         the destRouterID in the lowest order bytes.
   * @return The RouterBoundCtrlChTableEntry as a byte array with
   *         the destRouterID in the lowest order bytes.
   */
  public byte[] getBytes(){
    return entry;
  }

  /**
   * Returns a <code>String</code> representation of this entry
   * @return The <code>String</code> representation of this entry
   */
  public String toString() {
      return ("Server Flowid"+serverFlowId+"Destination Router ID: "
        + destRouterID.toString() +
        ", Next Hop: " + nextHop.toString()+",Goodness "+Goodness);
  }

  /**
   * Returns the length of this entry.
   * @return The length of this entry.
   */
```

```
     public short length(){
    try{
      return (short)entry.length;
    }catch(NullPointerException npe){
      return 0;
    }
  }
}
```

```
//-----------------------------------------
// Filename : Configuration.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project  : SAAM
//-----------------------------------------

package saam.message;

import saam.util.Array;
import saam.util.PrimitiveConversions;

 /**
   * Class to send values for setting up servers from demostation
   * for each server. By the help of this message server learns to have
   * as primary or backup, what flowid to use, which metric type to use
   * forming control channels.
   */
 public class Configuration extends Message{

   private byte serverType;//0-> main, 1->backup
   private int flowId;
   private byte metricType;
   private int cycleTime;
   private int globalTime;
   private byte[] bytes;

  /**
   * Constructs a Configuation message with the values provided.
   * @param byte sT value specifying server type.
   * @param int fid flow id of server.
   * @param byte mT metric type to use.
   * @param int cT cycle time to refresh the region continuosly
   * @param int gT value set the globaltimer at the first router.
   */
  public Configuration (byte sT,int fid,byte mT,int cT, .int gT){
    serverType = sT;
    flowId = fid;
    metricType = mT;
    cycleTime = cT;
    globalTime = gT;
    bytes = Array.concat(serverType,
            PrimitiveConversions.getBytes(flowId));
    bytes = Array.concat(bytes,metricType);
    bytes = Array.concat(bytes,
            PrimitiveConversions.getBytes(cycleTime));
    bytes = Array.concat(bytes,
            PrimitiveConversions.getBytes(globalTime));
  }

   /**
   * Constructs a Configuation message with the byte array provided.
   * @param byte [] ent byte array repreesenting configuration message.
   */
```

```
public Configuration (byte[] ent){
  bytes = ent;
  serverType = bytes[0];
  flowId = PrimitiveConversions.getInt( Array.getSubArray(ent,1,5));
  metricType = ent[5];
  cycleTime = PrimitiveConversions.getInt(
                  Array.getSubArray(ent,6,10));
  globalTime = PrimitiveConversions.getInt(
                  Array.getSubArray(ent,10,14));

}

 /**
 * method to return the server type.
 * @return int value.
 */
public byte getServerType(){
  return serverType;
}

/**
 * Returns the flowid of server.
 * @return int value
 */
public int getFlowId(){
  return flowId;
}

/**
 * Returns the metric type
 * @return byte value
 */
public byte getmetricType(){
  return metricType;
}

/**
 * returns the cycle time value
 * @param int value
 */
public int getCycleTime(){
  return cycleTime;
}

/**
 * returns the global time value
 * @param int value
 */
public int getGlobalTime(){
  return globalTime;
}

/**
 * Returns The byte array representation of this Message.
```

```
     * @return The byte array representation of this Message.
     */
    public byte[] getBytes(){
      return bytes;
    }

    /**
     * Returns The lenght of message .
     * @return short value .
     */

    public short length(){
      try{
        return (short)bytes.length;
      }catch(NullPointerException npe){
        return 0;
      }
    }

    /**
     * Returns a <code>String</code> representation of this Message.
     * @return The <code>String</code> representation of this Message
     */
    public String toString(){
      return("FlowId "+flowId+", metrictype "+metricType+" CycleTime
             "+cycleTime+" GlobalTime "+globalTime);
    }
  }
```

```
//-----------------------------------------
// Filename : TimeScale.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project  : SAAM
//-----------------------------------------

package saam.message;

import saam.util.Array;
import saam.util.PrimitiveConversions;

/**
* Class to scale running time of time specific values to platforms
*currently used.
*/
 public class TimeScale extends Message{

  private int scale;
  private byte[] bytes;

  /**
   * Constructs a TimeScale with the values provided.
   * @param int sc timescale value to use.
   */
  public TimeScale (int sc){
    scale = sc;
    bytes = PrimitiveConversions.getBytes(scale);
  }

  /**
   * Constructs a TimeScale message with the byte array provided.
   * @param byte [] byte byte array repreesenting   Timescale message.
   */
   public TimeScale (byte [] scl){
    bytes = scl;
    scale = PrimitiveConversions.getInt(Array.getSubArray(scl,0,4));
  }

  /**
   * Returns the scale value.
   * @return int value.
   */
  public int getTimeScale(){
    return scale;
  }

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
   */
  public byte[] getBytes(){
    return bytes;
  }
```

```
  /**
     * Returns the length of this entry.
     * @return The length of this entry.
     */
  public short length(){
    try{
      return (short)bytes.length;
    }catch(NullPointerException npe){
      return 0;
    }
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return("Time scale is  "+scale);
  }
} //end of TimeScale
```

```
//----------------------------------------
// Filename : ServerAgentSymetric.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project   : SAAM
//----------------------------------------

package saam.residentagent.server;

import saam.control.*;
import saam.residentagent.*;

import saam.server.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;

public class ServerAgentSymetric implements ResidentAgent,
MessageProcessor{

private SAAMRouterGui gui;
private ControlExecutive controlExec;
private Server myServer;


private String[] messageTypes = {"saam.message.Hello",
     "saam.message.FlowRequest",
     "saam.message.LinkStateAdvertisement",
     "saam.message.Configuration",    };

  public void install(ControlExecutive controlExec){
    gui=new SAAMRouterGui("ServerAgentSymetric");
    this.controlExec=controlExec;
    controlExec.registerMessageProcessor(this);
    myServer = new Server("classObject", controlExec);
//    myServer = new Server("database", controlExec);

    gui.sendText("\nCalling My Server method: autoconfig()");
    myServer.autoConfig();
  }

  public void processMessage(Message message){
    try{
      if(message instanceof Hello){
        gui.sendText("Received Message: "+((Hello)message));
        gui.sendText("Calling Server method: processHello()");
        myServer.processHello((Hello)message);
      }else if(message instanceof FlowRequest){
        FlowRequest request = (FlowRequest)message;
        gui.sendText("Received Message: "+ request);
        gui.sendText(
          "Calling Server method: processFlowRequest()");
        myServer.processFlowRequest((FlowRequest)message);

      }else if(message instanceof LinkStateAdvertisement){
```

```
        gui.sendText("Received Message: "+
          ((LinkStateAdvertisement)message));
        gui.sendText("Calling Server method: processLSA()");
        myServer.processLSA(
          (LinkStateAdvertisement)message);
      }
      //below adde by akkoc
      else if(message instanceof Configuration){
        gui.sendText("Received Message: "+
          ((Configuration)message));
        gui.sendText("Calling Server method: processConfiguration()");
        myServer.processConfiguration((Configuration)message);
      }

    }catch(Exception e){message = null;}
  }

  public String[] getMessageTypes(){
    gui.sendText("Server queried my message types");
  //  gui.sendText("Sending: "+messageTypes[0]);
    return messageTypes;
  }
  public String toString() {
    String it = "ServerAgentSymetric listening for: ";
    for(int i=0;i<messageTypes.length;i++){
      it += "\n" + messageTypes[i];
    }
    return it;
  }//toString()

  //the following methods are stubbed out as they are not used.
  public void uninstall(){
  }
  public Message query(Message message){
    return message;
  }
  public void transferState(ResidentAgent replacement){
  }
  public void receiveState(Message message){
  }
  public void receiveFlowResponse(FlowResponse flowResponse){
  }
  public void receiveEvent(SaamEvent event){
  }
}
```

```
//----------------------------------------
// Filename : ServerTbale.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project  : SAAM
//----------------------------------------

package saam.router;

import java.util.Hashtable;
import java.util.Vector;
import java.util.Enumeration;
import java.net.UnknownHostException;

import saam.control.*;
import saam.net.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;

import saam.residentagent.*;


/**
 * The <em>Server Table</em> stores the server flow id and server id
that
 * router listens to in the SAAM region
 */
public class ServerTable implements TableResidentAgent,MessageProcessor
{

  private TableGui gui;
  private Vector index= new Vector();
  private Vector names = new Vector();
  private ControlExecutive controlExec;
  private String[] messageTypes =
    {"saam.message.RouterBoundCtrlChTableEntry"};

  private Vector serverEntries= new Vector();

  /**
   * The constructor creates a ServerTable .
       */
      public ServerTable(ControlExecutive ce){
    names.add("FlowId");
    names.add("RBCCT");
    names.add("ServerId");
    int[] columnWidths = {50,150,250};
    gui = new TableGui(toString(), names, columnWidths);
    controlExec = ce;
    controlExec.registerMessageProcessor(this);
  }

  /**
```

```
   * Adds a new entry to the <em>Server Table</em>.
   * @param entry The entry to be added to the table.
   */

  public synchronized void add(ServerTableEntry entry){
    serverEntries.add(entry);
    gui.fillTable(getTable());
 }

  /**
       * Returns the message  types processed by the class
   * @return String array represantation of message name.
       */
  public String[] getMessageTypes(){
   return messageTypes;
  }

      /**
       * Removes an entry from the <em>server table</em>.
   * @param entry The entry to be removed from the table.
       */
      public void remove(ServerTableEntry entry){
    serverEntries.remove(entry);
    gui.fillTable(getTable());
      }


  public Message query(Message message){
    return null;
  }

      /**
       * Retrieves an ServerTableEntry from the table.
       * @param fid server id to be used as the lookup key.
       * @return The ServerTableEntry associated with flowid.
       */
      public synchronized ServerTableEntry getEntryByFlowId(int fid){
    Enumeration ei = serverEntries.elements();
    while(ei.hasMoreElements()){
      ServerTableEntry entry = (ServerTableEntry) ei.nextElement();
      if(entry.getFlowId() == fid){
        return entry;
      }
    }
    return  null;
      }

  /**
       * Method to check whether server with specified flow id is in
table.
       * @param fid server flow id to be used as the lookup key.
       * @return boolean value.
       */
  public boolean hasServer(int fid){
    Enumeration en = serverEntries.elements();
```

```
    while(en.hasMoreElements()){
      ServerTableEntry entry = (ServerTableEntry) en.nextElement();
      if(entry.getFlowId() == fid){
        return true;
      }
    }//end of while
    return  false;
      }

  /**
       * Method to check whether server with specified flow id and
server id
   * is in table.
       * @param fid server flow id to be used as the lookup key.
   * @param des id os server.
       * @return boolean value.
       */
  public boolean hasEntryForDestination (int fid, IPv6Address des){

    Enumeration en = serverEntries.elements();
    while(en.hasMoreElements()){
      ServerTableEntry entry = (ServerTableEntry) en.nextElement();
      RouterBoundCtrlChTable  tb = entry.getRouterBoundCtrlChTable();
      if(tb.hasEntry(des)){
        return true;
      }
    }//end of while

    return  false;
      }

   /**
       * Method to process servertable entry message
   * @param mes servertable entry message to be processed.
   * @return void.
       */
  public void processMessage(Message message){
    RouterBoundCtrlChTableEntry entry =
(RouterBoundCtrlChTableEntry)message;
    //first find the server to add by searching via flowId
    Enumeration e = serverEntries.elements();
    while(e.hasMoreElements()){
      ServerTableEntry ten = (ServerTableEntry) e.nextElement();
      if(ten.getFlowId() == entry.getServerFlowId()){
        ten.getRouterBoundCtrlChTable().add(entry);
      }

    }//end of while

  }  //end of process message

   /**
       * Method to check and remove RBCCT asociated with that server
   * so that routers dead or leaving the SAAM region no more displayed
   * @param fidOfSer server flow id.
```

```
  * @param lsq  last sequence number processed.
  * @return void.
      */
 public synchronized void refreshServerTable(int fidOfSer, int lsq){

   Enumeration e = serverEntries.elements();
   while(e.hasMoreElements()){
     ServerTableEntry ten = (ServerTableEntry) e.nextElement();
     if(ten.getFlowId() == fidOfSer ){
       RouterBoundCtrlChTable tb = ten.getRouterBoundCtrlChTable();
       tb.refreshRouterBoundControlTable(lsq);
     }
     break;

  }//end of while

}// end of  refreshServerTable


 /**
  * Returns the entire contents of this RouterBoundCtrlChTable
  * or null if this table is empty.
  * @return An array of all router-bound control channel entries
  *         currently in the table.
  */
 public Vector getTable(){

   if(serverEntries.isEmpty()) return null;
   Vector table = new Vector(serverEntries.size());

   Enumeration e = serverEntries.elements();
   while(e.hasMoreElements()){
      Vector oneRow = new Vector();
      ServerTableEntry entry = (ServerTableEntry) e.nextElement();

      oneRow.add(""+entry.getFlowId());
      oneRow.add(" RBCCT");
      oneRow.add(""+entry.getServerAddress());

      table.add(oneRow);
   }//while
   return table;
 }//getTable()


  public void uninstall(){
   serverEntries.clear();
 }


 /**
  * Method to get a server table enrty by as of server used as a key
  * @param ad id of server
  * @return ServerTableEntry corresponding to key.
```

```
   */ public synchronized ServerTableEntry
getEntryByServerId(IPv6Address ad){

    Enumeration el = serverEntries.elements();
    while(el.hasMoreElements()){
       ServerTableEntry look =  (ServerTableEntry) el.nextElement();
       IPv6Address address = look.getServerAddress();
       if(address.equals(ad)){
       //considered equal when server addresses are equal;
         return look;
       }

    }
    return null;
  }

  public void receiveState(Message message){

  }


 public void receiveFlowResponse(FlowResponse flowResponse){}

 public void transferState(ResidentAgent replacement){}

  public void install(ControlExecutive controlExec){}
 public void receiveEvent(SaamEvent se){}


  /**
   * Returns the String representation of this Object.
   * @return The String representation of this Object.
   */
  public String toString() {
      return "ServerTable";
  }//toString()
}
```

```
//-----------------------------------------
// Filename : ServerTbale Entry.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project  : SAAM
//-----------------------------------------

package saam.router;

import saam.net.IPv6Address;
import saam.router.RouterBoundCtrlChTable;
import saam.util.*;
import java.net.UnknownHostException;

/**
 * A RouterBoundCtrlChTableEntry object contains the destination router
 * ID (IPv6 address), and an IPv6 address representing the next hop.<p>
 * The entry could be either an add or a remove depending on which
 * constructor is used.  An add is required to have all fields, whereas
 * a remove only requires the field that is used as the lookup key in
 * the associated table.
 * Entries can be retrieved as a byte array using the getBytes()
 * method.
 */
public class ServerTableEntry {


    private int flowId;
    private RouterBoundCtrlChTable Table;
    private IPv6Address serverAddress;

    /**
     * Constructs an entry for the <em>Router-bound Control
     * Channel Table</em>.
     * @param destRouterID The IPv6 address to be stored.
     */
    public ServerTableEntry(int fid, RouterBoundCtrlChTable table,
IPv6Address srvId ) {
      flowId = fid;
      Table = table;
      serverAddress = srvId;

    }

    /**
     * Returns the next hop as an IPv6Address object.
     * @return The next hop as an IPv6Address object.
     */
    public IPv6Address getServerAddress(){
      return serverAddress;
    }

  /**
  * Returns the flow id of server.
  * @return int value.
```

```
  */
  public int getFlowId(){
    return flowId;
  }
 /**
  * Returns RouterBoundCtrlChTable  object.
  * @return TRouterBoundCtrlChTable object.
  */
  public RouterBoundCtrlChTable getRouterBoundCtrlChTable (){
    return Table;
  }

   /**
     * Returns a <code>String</code> representation of this entry
     * @return The <code>String</code> representation of this entry
     */
    public String toString() {
        return ("FlowId " + flowId+"ServerId " +
serverAddress.toString());
    }

}
```

```
//-----------------------------------------
// Filename : RouterBoundCtrlChTable.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project  : SAAM
//-----------------------------------------

package saam.router;

import java.util.Hashtable;
import java.util.Vector;
import java.util.Enumeration;
import java.net.UnknownHostException;
import saam.control.*;
import saam.net.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;
import saam.residentagent.*;

/**
 * The <em>Router-bound Control Channel Table</em> stores the
 * next hop IPv6 addresses for server-emitted control traffic
 * to reach a given destination router ID.<p>
 */
public class RouterBoundCtrlChTable extends Hashtable
       implements TableResidentAgent {

  private TableGui gui;
  private int flowIdOfServer;
  private Vector index= new Vector();
  private Vector names = new Vector();
  private ControlExecutive controlExec;

  /**
   * The constructor creates an RouterBoundCtrlChTable with
   * initial capacity of capacity.  Initial capacity should be a
       * prime number to help distribute the entries evenly
       * among the hash buckets.
       */
  public RouterBoundCtrlChTable(int flowS,int
                capacity,ControlExecutive ce){
    super(capacity, 0.5f);
    flowIdOfServer = flowS;
    names.add("Destination Router ID");
    names.add("Next hop IPv6 Address");
    names.add("Goodness");
    int[] columnWidths = {190,190,70};
    gui = new TableGui(toString(), names, columnWidths);
    this.controlExec=ce;
  }

  /**
   * Adds a new entry in the <em>router-bound control channel
table</em>.
```

```
   * @param entry The entry to be added to the table.
   */

  public synchronized void add(RouterBoundCtrlChTableEntry entry){
     IPv6Address destRouterID = entry.getDestRouterID();
     String destID =  new String(destRouterID.toString());
     put(destID,entry);
     gui.fillTable(getTable());

  }

  /**
   * Returns flowid of server
   * @return int value.
   */
  public int getFlowIdOfServer(){
     return flowIdOfServer;
  }

/**
 * Removes an entry from the <em>router-bound control table</em>.
 * @param entry The entry to be removed from the table.
*/
public synchronized void remove(RouterBoundCtrlChTableEntry entry){
  String destID = entry.getDestRouterID().toString();
  remove(destID);
  gui.fillTable(getTable());
}

/**
 * Retrieves an RouterBoundCtrlChTableEntry from the table.
 * @param destID The router ID to be used as the lookup key.
 * @return The RouterBoundCtrlChTableEntry associated with destID.
 */
public synchronized RouterBoundCtrlChTableEntry get(IPv6Address
destID){
      return (RouterBoundCtrlChTableEntry) get(destID.toString());
}


/**
  * Returns the entire contents of this RouterBoundCtrlChTable
  * or null if this table is empty.
  * @return An array of all router-bound control channel entries
  *         currently in the table.
  */
  public Vector getTable(){

    if(isEmpty()) return null;
    Vector table = new Vector(size());

    Enumeration e = elements();
    while(e.hasMoreElements()){
       Vector oneRow = new Vector();
       RouterBoundCtrlChTableEntry entry =
```

```
                        (RouterBoundCtrlChTableEntry) e.nextElement();

      oneRow.add(""+entry.getDestRouterID().toString());
      oneRow.add(""+entry.getNextHop().toString());
      oneRow.add(""+entry.getGoodness());

      table.add(oneRow);
   }//while
   return table;
}//getTable()

//since table resident agent extends resident agents following

public void uninstall(){
  clear();
}

public Message query(Message message){
  RouterBoundCtrlChTableEntry entry =
                        (RouterBoundCtrlChTableEntry)message;
  IPv6Address destRouterID = entry.getDestRouterID();
  String destID =  new String(destRouterID.toString());
  RouterBoundCtrlChTableEntry look =
              (RouterBoundCtrlChTableEntry)get(destID);
  return look;
}
public void receiveState(Message message){
  add(( RouterBoundCtrlChTableEntry)message);
}
/**
* Check RBCCT to romove entries that have old goodness value.
*Currently a difference vale greater than 1 is used as a contidion
* for removal
* @param entry The entry to be added to the table.
*/
public synchronized void refreshRouterBoundControlTable(int lastSQ){

  Enumeration et = elements();
  while(et.hasMoreElements()){
    RouterBoundCtrlChTableEntry entry = (RouterBoundCtrlChTableEntry)
                                                    et.nextElement();
    if((lastSQ - entry.getGoodness())>=2){
      remove(entry);
    }//end if
   }//end while
}//end refresh


/**
 * Determine whether an entry is in tablke or not.
 * @retrun boolean value
 */
public boolean hasEntry(IPv6Address ds){
  Enumeration et = elements();
  while(et.hasMoreElements()){
```

```
      RouterBoundCtrlChTableEntry entry =(RouterBoundCtrlChTableEntry)
                                                        et.nextElement();
      if( this.contains(ds)){
         return true;
      }//end if
    }//end while
    return false;
  }

 public void receiveFlowResponse(FlowResponse flowResponse){}

 public void transferState(ResidentAgent replacement){}

 public void install(ControlExecutive controlExec){}
 public void receiveEvent(SaamEvent se){}

  /**
   * Returns the String representation of this Object.
   * @return The String representation of this Object.
   */
  public String toString() {
      return "Router-bound Control Channel Table ("+flowIdOfServer+")";
  }//toString()
}
```

APPENDIX E. CLASSES MODIFIED IN SAAM FOR SCCP DEPLOYMENT

```
//-----------------------------------------
// Filename : ControlExeccutive.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project   : SAAM
//-----------------------------------------------------------------------


package saam.control;

import java.net.UnknownHostException;
import java.net.InetAddress;
import java.util.*;
import java.lang.*;

import saam.Translator;
import saam.router.*;
import saam.net.*;
import saam.message.*;
import saam.residentagent.*;
import saam.residentagent.router.*;

import saam.event.*;
import saam.util.SAAMRouterGui;
import saam.util.Array;

import saam.util.PrimitiveConversions;
import saam.EmulationTable;

public class ControlExecutive
  implements MessageProcessor, SaamTalker, SaamListener{

  //some well-known UDP ports
  public static final int ECHO_PORT          = 7;
  public static final int DISCARD_PORT       = 9;
  public static final int DAYTIME_PORT       = 13;
  public static final int TIME_SERVER_PORT   = 37;
  public static final int DNS_PORT           = 53;
  public static final int WWW_HTTP_PORT      = 80;
  public static final int CHAT_PORT          = 531;

  //saam ports/channels
  public static final int HIGHEST_WELL_KNOWN_PORT       = 1023;
  public static final int MAX_PORT                      = 65531;
  public static final int SAAM_CONTROL_PORT
            = 8000;
  public static final int ROUTER_STATUS_CHANNEL
      = 80000;



  /**
 * The ControlExecutive registers with itself as a MessageProcessor
```

```
* capable of
* processing messages of the following types.
*/
 private static final String[] messageTypes =
   {"saam.message.InterfaceID",
    "saam.message.ServerID",
    "saam.message.FlowResponse",
    "saam.message.DemoHello",
    "saam.message.DCM",
    "saam.message.UCM",
    "saam.message.ParentNotification",
    "saam.message.TimeScale",
  };

 private static final boolean ROUTER_UP = true;
 private static final boolean ROUTER_DOWN = false;


 /**
  * The initial status of the router is false.  As key router
  * are added, this status is updated to reflect the router's ability
  * to route packets.
  */
 private boolean routerStatus = ROUTER_DOWN;

 /*
 * The following boolean variables represent the status of the
 * elements necessary  to stand up a router.
 */
 private boolean helloMessageReceived;
 private boolean arpCacheReady;
 private boolean flowRoutingTableReady;
 private boolean emulationTableReady;
 private boolean outboundInterfaceReady;

 private int interfaceCount;
 private int numberOfSchedulersPresent;
 private int nextInboundInterface;
 private SAAMRouterGui gui;
 private MainGui mainGui;
 private PacketFactory packetFactory;

 private TransportInterface transportInterface;
 private ResidentAgent arpCache;

 private static RoutingAlgorithm routingAlgorithm;

 private Interface currentInterface;

 private Object theLock= new Object();//critical section lock

 //below are added by akkoc
 private AutoConfigurationExecutive autoConExec;
 private int timeScale;
 private static RouterBoundCtrlChTable rotBonConTable;
```

```
  private IPv6Address routerId= new IPv6Address();
  private static EmulationTable emTable;
  private boolean isServer = false;
  private ServerTable serverTable;

  /**
   * If a ServerID Message comes from the DemoStation,
   * the IPv6Address associated with that ServerID will
   * be set as the dest in the IPv6Header of packets sent out
   * on Server-bound flow, otherwise, the default IPv6Address
   * will be set as the dest.
   */
  // private IPv6Address serverIP = new IPv6Address();  //Akkoc removed

  /**
   * The ServerID will be sent by the DemoStation.  //Akkoc removed
   */
//  private ServerID serverID;   //Akkoc removed

  /**
   * The Vector that contains Interfaces that have been instantiated on
   * this router.  Default size = 4.
   */
  private Vector interfaces = new Vector(4);

  /**
   * The Vector of IDs for each interface that has been instantiated on
   * this router.  Default size = 4.
   */
  private Vector interfaceIDs = new Vector(4);

  /**
   * The Vector of talkers that have passed the registration process
   * are authorized to talk on channels.
   */
  private Vector activeTalkers = new Vector();

  /**
   * The Hashtable of ResidentAgents that have been instantiated by the
   * ControlExecutive.
   */
  private Hashtable agents = new Hashtable();

  /**
   * The Hashtable of ResidentAgentCustomers that have registered to
   * receive ResidentAgent replacements as they arrive.
   */
  private Hashtable agentCustomers = new Hashtable();

  /**
   * The Hashtable of MessageProcessors that have registered with this
   * ControlExecutive.
   */
  private Hashtable messageProcessors =
    new Hashtable();
```

```
  /**
   * The Hashtable of Channels that have been instantiated by CE.
   */
  private Hashtable activeChannels = new Hashtable();

  /**
   * The Hashtable of channels that a given SaamTalker is registered to
talk on.
   */
  private Hashtable channelsTalkerHas = new Hashtable();

  /**
   * The Hashtable of channels that a given SaamListener is registered
to listen on.
   */
  private Hashtable channelsListenerHas = new Hashtable();

  /**
   * The Hashtable of Objects that have requested flows.
   */
  private Hashtable flowRequestors = new Hashtable();

  /**
   * The Hashtable of Objects that have been assigned flows.
   */
  private Hashtable assignedFlows = new Hashtable();


  /**
   * Instantiates and sets up communication with all Objects that are
necessary
   * to allow the ControlExecutive to start receiving ResidentAgents
and Messages.
   */
  public ControlExecutive(){
    mainGui = new MainGui(this,"SAAM Router Prototype");
    gui = new SAAMRouterGui(toString());
    transportInterface =  new TransportInterface(this);
    //for receiving inbound packets
    packetFactory = new PacketFactory(this);

    emTable =new EmulationTable();
    arpCache = new ARPCache();
    serverTable = new ServerTable(this);
    autoConExec = new AutoConfigurationExecutive(this);

    //this should eventually become a ResidentAgent
    routingAlgorithm = new RoutingAlgorithm(this, arpCache);

    //Here is where the CE registers itself as a MessageProcessor
    registerMessageProcessor(this);

    /* Enable Talking on channel that Translator is listening on for
       router status updates.*/
```

```
  try{
    addTalkerToChannel(this, ROUTER_STATUS_CHANNEL);
  }catch(ChannelException ce){
    gui.sendText(ce.toString());
  }

  try{
    //Get ownership of the SAAM_CONTROL_PORT so other applications
    //cannot.  When the TransportInterface sees a packet destined
    //for this port, it will not forward the packet directly on
    //the SAAM_CONTROL_PORT, rather, it will forward the packet
    //to the PacketFactory on the
   ProtocolStackEvent.PACKETFACTORY_CHANNEL
    monitorPort(this, SAAM_CONTROL_PORT);
    gui.setTextField("Monitoring emulated port "+SAAM_CONTROL_PORT);
  }catch(PortAccessDeniedException pade){
    gui.sendText(pade.toString());
  }
  mainGui.updateDisplay();
}//ControlExecutive()

/**
 * Returns the IPv6Address of the server controlling this router
 * @return The IPv6Address of the server controlling this router.
 */
/*public IPv6Address getServerIP(){
  return serverIP;
} */ // akkoc removed

/**
 * Returns the Servertable maintained by the router
 * @return ServerTable containing the entries
 */
public synchronized ServerTable getServerTable(){
  return serverTable;
}

 /**
 * Returns the timesacle value for those requiring it
 * @return int timescale value
 */
public int getTimeScale(){
  return timeScale;
}
 /**
 * Returns the EmulationTable of the router
 * @return EmulationTable containing the entries
 */
public synchronized EmulationTable getEmulationTable(){
  return emTable;
}

/**
 * To let know whether to behave as router or server for classes
 * requiring that information
```

```
     * @return boolean value
     */
    public boolean getIsServer(){
       return isServer;
    }

    /**
     * Returns the Router idd for the router
     * @return IPv6Address of the router .
     */
    public IPv6Address getRouterId(){
      return this.routerId;
    }

    /**
     * Returns the status of the ARPCache.
     * @return The status of the ARPCache ResidentAgent. (if the ARPCache
     * contains an entry that corresponds to nexthop for Server-bounded
     * flow, this method returns true).
     */
    public boolean getArpCacheStatus(){
      return arpCacheReady;
    }

    /**
     * Returns the status of the EmulationTable.
     * @return The status of the EmulationTable.  (if the EmulationTable
     * contains an entry that corresponds to the nexthop for Server-bound
     * flow, this method returns true).
     */
    public boolean getEmulationTableStatus(){
      return emulationTableReady;
    }

    /**
     *There are three tables in every SAAM router:
     *ARPCache,EmulationTable,
     * and the FlowRoutingTable.  Each of these tables notifies the CE
     * when it is ready to serve the router.  When all of these tables
  are ready and
     * a few other conditions are met, the CE sends a notification to
     * the Translator.
     * @param o The Object sending the update.
     * @param status The status of o.
     */
    public void updateCoreServiceStatus(
      Object o, boolean status){

      String className = o.getClass().getName();
      if(className.equals(
        "saam.residentagent.router.ARPCache")){
         arpCacheReady = status;
         updateRouterStatus();
      }else if (className.equals("saam.Translator")){
         emulationTableReady = status;
```

```
        updateRouterStatus();
      }
      //do nothing if another Object called this method
    }

  // methods below are modified/killed by [akkoc]
    /**
     * The FlowRoutingTable uses this method to notify the CE when it
     * is ready to serve the router.
     * @param o The Object sending the update.
     * @param serverBoundNextHop The IPv6Address of nexthop
     * associated with Server-bound flow.
     */
  /* public void updateCoreServiceStatus(Object o, IPv6Address
serverBoundNextHop){

        String className = o.getClass().getName();
        this.serverBoundNextHop = serverBoundNextHop;
        boolean status = (serverBoundNextHop.equals(
            new IPv6Address()))? false:true;
        flowRoutingTableReady = status;
        //the following logic does not work for some reason...
        //I think it's the ArpCache.query method
        if(!arpCacheReady){
          if(routingAlgorithm.checkARPCache(
            new ARPCacheEntry(serverBoundNextHop))){
            arpCacheReady=true;
          }
        }
        updateRouterStatus();
    } */

    /**
     * Returns the IPv6Address representing the next hop associated with
     * Server-bound flow.
     * @return The IPv6Address representing the next hop associated with
     * Server-bound flow.
     */
  /* public IPv6Address getServerBoundNextHop(){
      return serverBoundNextHop;
    }   */

    /**
     * Performs a series of checks to determine the status of the router
     * then updates the status accordingly.
     */
    private synchronized void updateRouterStatus(){

    // used only to define whether router is up or not
    // displayRouterStatus();
     /* String myAddress = null;
      try{
        myAddress = InetAddress.getLocalHost().getHostAddress();
      }catch(UnknownHostException uhe){}    */
```

```
//compare local address with the address of the server.  If the
//two addresses are the same and there is at least one interface
//to process traffic, then the outbound interface for this server
//is ready.

//19Dec99[akkoc]  - logic belowhas been changed acc, to new paradigm
// if there is at leat one interfce ready then out bound interface is
//ready
//OLD
/* if(myAddress.equals(serverID.getIPv4()) &&
      (interfaces.size()>=1)){
      outboundInterfaceReady=true;
  }else if
//otherwise, compare the network portion of the IPv6Address of the next
//hop associated with Server-bound flow to the network portions of the
//IPv6Addresses of the interfaces on this router.  If there is a match,
//the outbound interface is ready.
(routingAlgorithm.determineOutboundInterface(interfaces,
      serverBoundNextHop)!=null){
      outboundInterfaceReady=true;
} */
//NEW

if(interfaces.size()>=1){
    outboundInterfaceReady=true;
}
//if all the conditions are met, notify the Translator that the router
//is ready.
if(helloMessageReceived && arpCacheReady && emulationTableReady &&
       outboundInterfaceReady){

    if(routerStatus==ROUTER_DOWN){
    //need to bring router status up since it satisfies all criterias
      routerStatus=ROUTER_UP;
    //notifying Translator
      RouterStatusEvent  event = new RouterStatusEvent(toString(),this,
                                 ROUTER_STATUS_CHANNEL,routerStatus);
        try{
          talk(event);
        }catch(ChannelException ce){
          gui.sendText(ce.toString());
        }

        gui.sendText("Router is still UP NOW...");
      }else{
      routerStatus=ROUTER_DOWN;
      //bringRouterDown();
      gui.sendText("Router is still down...");
     }
   }
  }// end of method

  /**
   * Displays the current status of the router.
   */
```

```
  private void displayRouterStatus(){
    gui.sendText("\nCurrent Router Status:");
    gui.sendText(" helloMessageReceived:   "+helloMessageReceived);
    gui.sendText(" arpCacheReady:          "+arpCacheReady);
    gui.sendText(" flowRoutingTableReady:  "+
      flowRoutingTableReady);
    gui.sendText(" emulationTableReady:    "+emulationTableReady);
    gui.sendText(" outboundInterfaceReady: "+
      outboundInterfaceReady+
      "\n");
  }

  /**
   * In the SAAM architecture, traffic cannot be sent between hosts if
   * hosts are not assigned flows.  This method provides the mechanism
   * which hosts request flows from the SAAM server.  With the
information
   * provided in the parameters, this method constructs a
saam.message.FlowRequest.
   * It then sends that FlowRequest to the protocol stack for
transmission to
   * the SAAM server.  Note: If the requestor is not listening to a
port, the
   * requestor should first call the listenToRandomPort method for a
port assignment.
   * @param requestor The Object requesting the flow.
   * @param sourcePort The local port that requestor is listening on.
   * @param destHost The IPv6Address of the destination.
   * @param requestedDelay The amount of delay the requestor will
tolerate.
   * @param requestedLossRate The rate of loss the requestor will
tolerate.
   * @param requestedThroughput The amount of throughput the requestor
will tolerate.
   */
  public synchronized long requestFlow(ResidentAgent requestor,
   short sourcePort, IPv6Address destHost, int requestedDelay,
   int requestedLossRate, int requestedThroughput)
     throws FlowException{

      long timeStamp = System.currentTimeMillis();
      flowRequestors.put(new Long(timeStamp),requestor);
      IPv6Address sourceHost =
        ((InterfaceID)interfaceIDs.get(0)).getIPv6();
      FlowRequest request = new FlowRequest(
                            sourceHost,destHost,timeStamp,
       requestedDelay,requestedLossRate,requestedThroughput);

      //FlowRequests travel on Server-bound flow.
      short destPort = (short)SAAM_CONTROL_PORT;
    // below needs modification after Autoconfiguration is completed
    /*  try{
      //  send(this, request, MainServerBoundCtrlFlowID,
                              sourcePort,serverIP,destPort);
      }catch(FlowException fe){
```

```
        gui.sendText(fe.toString());
      }    */
      return timeStamp;
  }

  /**
   * Objects that have been assigned flows can send Messages with this
method.
   * In order to use this method, Objects must first request a flow
using
   * requestFlow method; and then receive a flow assignment from
server
   * @param sender The Object sending the message.
   * @param message The subclass of saam.message.Message to be sent.
   * @param flowID ID that has been assigned for traffic from sender
   *        destined for destHost.
   * @param sourcePort The port on the local machine that sender is
listening on.
   * @param destHost The IPv6Address of the destination.
   * @param destPort The port to which destHost is listening.
   */
  public synchronized void send(Object sender, Message message,
    int flowID, short sourcePort, IPv6Address destHost,
    short destPort) throws FlowException{

      if(sender.getClass().getName().equals(
        "saam.residentagent.router.LsaGenerator")){
          sender = this;
      }
      gui.sendText("Sending Message...");
      PacketFactory packetFactory = new PacketFactory();

         packetFactory.append(message);

         SAAMPacket saamPacket = null;
        try{
          saamPacket = new SAAMPacket(
          packetFactory.getBytes());
        }catch(UnknownHostException uhe){
          throw new FlowException(sender+
           " Problem building packet "+flowID);
        }

      //call the send method that takes an IPv6Packet,
      //using the IPv6Packet constructed by the TransportInterface
      IPv6Packet v6Packet = transportInterface.buildIPv6Packet(sender,
        saamPacket,flowID,sourcePort,destHost,destPort);
      gui.sendText(">> Message payload size = " +
                      v6Packet.getPayload().length);
      try{
        saamPacket = new SAAMPacket(v6Packet.getPayload());
      }catch(UnknownHostException uhe){
        throw new FlowException(sender+
          " Problem building packet "+flowID);
      }
```

```
      send(sender,v6Packet);
  }

  /**
   * Objects that have been assigned flows can send DCM messages with
this method.
   * @param sender The Object sending the message.
   * @param message The DCM message to send
   * @param flowID The flow ID that has been assigned for traffic from
sender
   *        destined for destHost.
   * @param sourcePort The port on the local machine that sender is
listening on.
   * @param destHost The IPv6Address of the destination.
   * @param destPort The port to which destHost is listening.
   */
  public synchronized void sendDCM(Object sender,
                        DCM message, int flowID,
                        short sourcePort, IPv6Address destHost,
                        short destPort)
throws FlowException{

    PacketFactory packetFactory = new PacketFactory();
    packetFactory.appendDCM(message);
    SAAMPacket saamPacket = null;
    long time = System.currentTimeMillis();
    byte numMessages= 1;
    SAAMHeader saamHeader = new SAAMHeader(time,numMessages);
    try{
      saamPacket = new SAAMPacket(
                      saamHeader, packetFactory.getDCMBytes());
    }catch(Exception uhe){
        throw new FlowException(sender+
          " Problem building packet "+flowID);
    }
    //call the send method that takes an IPv6Packet,
    //using the IPv6Packet constructed by the TransportInterface
    IPv6Packet v6Packet = transportInterface.buildIPv6Packet(
               Sender,saamPacket,flowID,sourcePort,
               destHost,destPort);
    gui.sendText(">> DCM Message payload size = " +
                       v6Packet.getPayload().length);
    gui.sendText(">> DCM Message IN IPV6 from size is = " +
                        v6Packet.getBytes().length);

    send(sender,v6Packet);


  } //end of sendDCM

  /**
   * Objects that have been assigned flows can send UCM with this
method.
   * @param sender The Object sending the message.
   * @param message The UCM message to send
```

```
 * @param flowID The flow ID assigned for traffic from sender
 *        destined for destHost.
 * @param sourcePort The port on machine that sender is listening on.
 * @param destHost The IPv6Address of the destination.
 * @param destPort The port to which destHost is listening.
 */

public synchronized void sendUCM(Object sender,
                         UCM message, int flowID,
                         short sourcePort,IPv6Address destHost,
                         short destPort)
                         throws FlowException{
  PacketFactory packetFactory = new PacketFactory();
  packetFactory.appendUCM(message);
  SAAMPacket saamPacket = null;
  long time = System.currentTimeMillis();
  byte numMessages= 1;
  SAAMHeader saamHeader = new SAAMHeader(time,numMessages);
  try{
    saamPacket = new SAAMPacket(saamHeader,
                     packetFactory.getUCMBytes());
  }catch(Exception uhe){
      throw new FlowException(sender+
        " Problem building packet "+flowID);
  }
  //call the send method that takes an IPv6Packet,
  //using the IPv6Packet constructed by the TransportInterface
  IPv6Packet v6Packet = transportInterface.buildIPv6Packet(sender,
                     saamPacket,flowID,sourcePort,destHost,destPort);
  gui.sendText(">> UCM Message payload size = " +
                         v6Packet.getPayload().length);
  gui.sendText(">> UCM Message IN IPV6 from size is = " +
                         v6Packet.getBytes().length);
  send(sender,v6Packet);
} //end of sendUCM

/**
 * Objects that have been assigned flows can send PN via this method.
 * @param sender The Object sending the message.
 * @param message The PN message to send
 * @param flowID The flow ID assigned for traffic from sender
 *        destined for destHost.
 * @param sourcePort The port machine that sender is listening on.
 * @param destHost The IPv6Address of the destination.
 * @param destPort The port to which destHost is listening.
 */
public synchronized void sendPN(Object sender,
                         ParentNotification message,int flowID,
                         short sourcePort,IPv6Address destHost,
                         short destPort)
                         throws FlowException{

  PacketFactory packetFactory = new PacketFactory();
  packetFactory.appendPN(message);
  SAAMPacket saamPacket = null;
```

```
    long time = System.currentTimeMillis();
    byte numMessages= 1;
    SAAMHeader saamHeader = new SAAMHeader(time,numMessages);
    try{
      saamPacket = new SAAMPacket(saamHeader,
                                  packetFactory.getPNBytes());
    }catch(Exception uhe){
        throw new FlowException(sender+
          " Problem building packet "+flowID);
    }
    //call the send method that takes an IPv6Packet,
    //using the IPv6Packet constructed by the TransportInterface
    IPv6Packet v6Packet = transportInterface.buildIPv6Packet(sender,
                    saamPacket,flowID,sourcePort,destHost,destPort);
    gui.sendText(">> PN Message payload size = " +
                         v6Packet.getPayload().length);
    gui.sendText(">> PN Message IN IPV6 from size is = " +
                           v6Packet.getBytes().length);
    send(sender,v6Packet);
  }//end sendPN

  /**
   * Objects that assigned flows can send SAAMPackets wia this method.
   * To use this method, Objects must first request a flow using the
   * requestFlow method; and then receive a flow assignment from server
   * @param sender The Object sending the message.
   * @param saamPacket The SAAMPacket to be sent.
   * @param flowID The flow ID assigned for traffic from sender
   *        destined for destHost.
   * @param sourcePort The port on  that sender is listening on.
   * @param destHost The IPv6Address of the destination.
   * @param destPort The port to which destHost is listening.
   */
  public synchronized void send(Object sender, SAAMPacket saamPacket,
    int flowID, short sourcePort, IPv6Address destHost,
    short destPort) throws FlowException{

      //call the send method that takes an IPv6Packet,
      //using the IPv6Packet constructed by the TransportInterface
      gui.sendText("Sending SAAM Packet...");
      IPv6Packet v6Packet = transportInterface.buildIPv6Packet(sender,
        saamPacket,flowID,sourcePort,destHost,destPort);
      send(sender,v6Packet);
  }//send()

  /**
   * Objects assigned flows can send IPv6Packets with this method.
   * To use this method, Objects must first request a flow using the
   * requestFlow method; and then receive flow assignment from server.
   * note: If packet is destined to Interface that is one this router,
   * the packet will be delivered without flow id verification.
   * @param sender The Object sending the message.
   * @param ipv6Packet The IPv6Packet to be sent.
   */
  public synchronized void send(Object sender, IPv6Packet ipv6Packet)
```

```
    throws FlowException{

    int flowID = ipv6Packet.getHeader().getFlowLabel();
    //verify that the sender owns the flowID
   // gui.sendText("Sending IPv6 Packet on flow "+flowID);
    if(!routingAlgorithm.isApplicationLayerPacket(ipv6Packet)){
      ResidentAgent agent = (ResidentAgent)assignedFlows.get(
            new Integer(flowID));
      if(sender.getClass().getName().equals("saam.server.Server")
        || (agent!=null&&agent.equals(sender)) || sender.equals(this)){
        //Here we forward the packet to an inbound interface
        //so it will be processed just as if it were an inbound
        //packet.
        ProtocolStackEvent event = new ProtocolStackEvent(
         sender.toString(),this,
         ProtocolStackEvent.getFromNICToInterfaceChannel(
           nextInboundInterface),
            ipv6Packet.getBytes());
        try{
          gui.sendText("\n>> Enqueuing packet for transmission at
            channel" +
            event.getChannel_ID());
          gui.sendText("  >> nextInboundInterface = " +
                      nextInboundInterface+" flowid is "+flowID);
          talk(event);
        }catch(ChannelException ce){
          gui.sendText(ce.toString());
        }
       //routingAlgorithm.routeInboundPacket(ipv6Packet.getBytes());
        nextInboundInterface++;
        if(nextInboundInterface>=interfaces.size()){
          nextInboundInterface=0;
        }
      }else {
        gui.sendText(sender+
          " doesn't own flow "+flowID);
        throw new FlowException(sender+
          " doesn't own flow "+flowID);
      }
    }else{
      //packet is destined to an Interface on this router,to go outside
      //so we forward it to the TransportInterface for delivery
      //on the proper emulated UDP port.

      IPv6Header v6Header = ipv6Packet.getHeader();

if(v6Header.getSource().toString().equals(IPv6Address.DEFAULT_HOST)){

v6Header.setSource(((Interface)interfaces.get(0)).getID().getIPv6());
        ipv6Packet.setHeader(v6Header);
      }

      //gui.sendText("received app. layer packet from application
layer");
      gui.sendText("\nforwarding to TransportInterface");
```

```
        SAAMPacket saamPacket = null;
        try{
          saamPacket = new SAAMPacket(ipv6Packet.getPayload());


        }catch(UnknownHostException uhe){
          gui.sendText(uhe.toString());
        }
        ProtocolStackEvent event = new ProtocolStackEvent(
          sender.toString(),
          //here we trick the TransportInterface into thinking this
          //event came from the routingAlgorithm.  Also, since the
          //routingAlgorithm is already registered to talk on this
          //channel, we make it past the security check that ensures
          //the talker is registered on the channel.
          routingAlgorithm,ProtocolStackEvent.
            FROM_ROUTINGALGORITHM_TO_TRANSPORTINTERFACE_CHANNEL,
          ipv6Packet.getBytes());
        try{

          talk(event);

        }catch(ChannelException tde){
          gui.sendText(tde.toString());
        }

      }


    }//send()

  /**
  *To send a flowRequest or any other type of traffic,
  *sending Objects must be listening to a port.This method assigns
  Objects a
  * random port that is higher than the highest well-known port, but no
  higher than
  * MAX_PORT.
  * @param listener The SaamListener requesting a random port.
  */
  public int listenToRandomPort(SaamListener listener){
    int port = listenToRandomChannel(
    listener, HIGHEST_WELL_KNOWN_PORT+1,MAX_PORT);
    try{
  //the TransportInterface must be able to talk on the new port in
  //order to deliver traffic to the listener
    addTalkerToChannel(transportInterface, port);
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }
    return port;
    //also register the TransportInterface as a talker on
    //this port.
  }//listenToRandomPort()
```

```
/**
* Ports reside on Channels 0-MAX_PORT; any Channel higher than MAX_PORT
*is a Channel that is not associated with a port.  Examples of this are
* the communications Channels within the protocol stack.
* @param listener The SaamListener requesting a random channel.
* Note: Since this method is private, only the ControlExecutive can
* assign listeners to a random channel.
* @param lowestChannel The lowest channel in the range to be selected
from.
* @param highestChannel The highest channel in the range to be selected
from.
*/
private int listenToRandomChannel(
    SaamListener listener, int lowestChannel,
    int highestChannel){
    int channelFound = 0;
    boolean exception = true;
    while(exception){
      try{
        channelFound = (lowestChannel+(
          new Random()).nextInt(highestChannel-lowestChannel));
        addListenerToChannel(listener, channelFound);
        exception = false;
      }catch(Exception e){}
    }//while()
    return channelFound;
  }//listenToRandomChannel()

  /**
   * Returns the array of Strings representing the class
   * names of the messages this Object will register to process.
   * @return The array of Strings representing the class names
   * of the messages this Object will register to process.
   */
  public synchronized String[] getMessageTypes(){
    return messageTypes;
  }//getMessageTypes()

  /**
   * This private method is used by the ControlExecutive to perform the
   * steps necessary to instantiate an Interface.
   * @param id The InterfaceID that will be assigned to the interface
being
   * instantiated.  If an Interface with this id is already up, the
request
   * will be ignored.
   */
  private void standUpInterface(InterfaceID id){
    boolean alreadyActive = false;
    for (int i=0;i<interfaceIDs.size();i++){
      if(((InterfaceID)interfaceIDs.get(i)).equals(id)){
        alreadyActive = true;
        break;
      }
```

```
    }

    if(!alreadyActive){
      interfaceIDs.add(id);
      gui.sendText(" Instantiating interface["+interfaceCount++ +"]");
      gui.sendText(" IPv6Address: "+id.getIPv6().toString());

      Interface thisInterface = new Interface(this, id);
      interfaces.add(thisInterface);
      updateRouterStatus();
      routingAlgorithm.addInterface(thisInterface);
      try{
        addTalkerToChannel(this,
          ProtocolStackEvent.getFromNICToInterfaceChannel(
            interfaces.size()-1));
      }catch(ChannelException ce){
        gui.sendText(ce.toString());
      }
    }else{
      gui.sendText("Interface already active...");
    }

    //[Akkoc] added for router_id determination
    // to check whether this new interface can be routerId
    // Highest IPv6Adress has been taken as routerId
    byte[] candidate = id.getIPv6().getAddress();
    byte[] rId = routerId.getAddress();

      for(int i=0; i<rId.length; i++){
        if(candidate[i] < rId[i]) {
           break;
        }else if(candidate[i] > rId[i]) {
           routerId = id.getIPv6();
        }//end else if
      } // end for

      gui.sendText("Now router id is "+ getRouterId().toString());


  }//standUpInterface()

/**
* This method contains the logic needed by the ControlExecutive
* to process the Messages it is registered to process.
* @param message The subclass of saam.message.Message to be processed.
*/
public void processMessage(Message message){
  String name = message.getClass().getName();
  if(name.equals(messageTypes[0])){
    InterfaceID id = (InterfaceID)message;
    standUpInterface(id);
    updateRouterStatus();
  }else if(name.equals(messageTypes[1])){
    try{
    //  serverID = ((ServerID)message); // akkoc killed
```

```
    //  serverIP = serverID.getIPv6(); //  akkoc killed
    }catch(Exception e){
      gui.sendText("Error processing serverID: "+e.toString());
    }
  }else if(name.equals(messageTypes[2])){
    gui.sendText("Got a FlowResponse.. ");
    FlowResponse response = (FlowResponse)message;
    long timeStamp = response.getTimeStamp();
    gui.sendText("TimeStamp: "+timeStamp);
    int flowID = response.getFlowId();
    gui.sendText("flowID: "+flowID);
    ResidentAgent requestor =
      (ResidentAgent)flowRequestors.get(new Long(timeStamp));
    gui.sendText("Forwarding to Requestor: "+requestor);
    if(requestor!=null){
      assignedFlows.put(new Integer(flowID),requestor);
    requestor.receiveFlowResponse(response);
    }//else do nothing


    }else if(name.equals(messageTypes[3])){
      //received a saam.message.DemoHello from the DemoStation
      gui.sendText("received message type: " + messageTypes[3]);
      //
      Vector helloInterfaces = ((DemoHello)message).getInterfaceIDs();
      for(int i=0;i<helloInterfaces.size();i++){
        InterfaceID id = (InterfaceID)helloInterfaces.get(i);
        standUpInterface(id);
      }
      helloMessageReceived = true;  //DemoHello - CRC
      updateRouterStatus();
      mainGui.updateDisplay();
    }

    // below else cases are added by [akkoc]
    else if(name.equals(messageTypes[4])){
      gui.sendText("received message type: " + messageTypes[4]);
      DCM receivedDcm = (DCM)message;
      autoConExec.processDCM(receivedDcm);

    }//end of else if for DCM
    else if(name.equals(messageTypes[5])){
      gui.sendText("received message type: " + messageTypes[5]);
      UCM ucm =(UCM) message;
      autoConExec.processUCM(ucm);

    }//end of elseif for ucm
    else if(name.equals(messageTypes[6])){ //Parent Notification
      gui.sendText("\n received message type: " + messageTypes[6]);
      ParentNotification pn = (ParentNotification)message;
      autoConExec.processPN( pn );

    }//end of elseif for parent notification
    else if(name.equals(messageTypes[7])){ //Parent Notification
      gui.sendText("\n received message type: " + messageTypes[7]);
```

```
      TimeScale ts = (TimeScale)message;
      timeScale = ts.getTimeScale();

    }//end of else if for parent notification

  }//processMessage()


/**
* Returns the number of Interfaces that have been stood up by this CE.
* @return The number of Interfaces that have been stood up by this CE.
*/
public int getNumberOfInterfaces(){
  return interfaceCount;
}

/**
 * Makes access to RoutingAlgorithm  possible for requiring classes.
 * @return RoutingAlgorithm object
 */
 public synchronized RoutingAlgorithm getRoutingAlgorithm(){
    return  routingAlgorithm;
 }

/**
* Makes access to AutoConfigurationExec possible for requiring classes.
* @return AutoConfigurationExecutive  object
*/
public AutoConfigurationExecutive getAutoConfigurationExecutive(){
  return  autoConExec;
}

/**
* This is the method MesProcessors use to register to process Messages.
* The ContExec retrieves the list of Messages from mp by calling
* mp.getMessageTypes().
* @param mp The MessageProcessor that is registering with this ConExec.
*/
public void registerMessageProcessor(MessageProcessor mp){
    gui.sendText("Registering MessageProcessor: "+mp);
    Object[] elementsIProcess = null;
    //retrieve the list of Messages
    elementsIProcess = mp.getMessageTypes();
    for(int i=0;i<elementsIProcess.length;i++){
      String element = (String)elementsIProcess[i];
      if(!element.equals("saam.message.FlowResponse")){
        MessageProcessor oldProcessor =
          (MessageProcessor)messageProcessors.put(element,mp);
         // notify old Processor that it will no longer
         // receive this type of message.
        }else{
          if(mp.equals(this)){
            messageProcessors.put(element,this);
          }else{
            gui.sendText("DENIED: "+element);
```

```
            }
          }
        }
    }

  /**
 * ResidentAgentCustomers use this method to register to receive
ResidentAgent
 * updates from the ControlExecutive when they arrive.  If an agent is
replaced
 * with a new agent, the ControlExecutive will call the customer's
replaceAgent
 * method.
 * @param rac the ResidentAgentCustomer requesting registration.
 */
public void registerCustomer(ResidentAgentCustomer rac){
    Object[] agentsIUse = null;
    agentsIUse = rac.getAgentTypes();
    for(int i=0;i<agentsIUse.length;i++){
      String agent = (String)agentsIUse[i];
      Vector customers = null;
      synchronized(agentCustomers){
        customers = (Vector)agentCustomers.get(agent);
      }
      if(customers == null){
        customers = new Vector();
        agentCustomers.put(agent,customers);
      }
      customers.add(rac);
     // oldProcessor.
    }
}

/**
 * Replaces an existing ResidentAgent with an incoming ResidentAgent
 * of the same class name.  If there are multiple instances of the
existing
 * agent, the replacement will occur instance for instance.
 * @param classObject The Class of the incoming agent.
 * @param className The class name of incoming ResidentAgent subclass
 */
private void replaceOldAgent(Class classObject, String className){

    boolean badAgent = false;
    Vector agentInstances = new Vector();
    int numberOfInstancesNeeded = 1;
    if(className.equals("saam.residentagent.router.Scheduler")){
      synchronized(interfaces){
        numberOfInstancesNeeded = interfaces.size();
      }
    }
    for (int i=0;i<numberOfInstancesNeeded;i++){
      try{
//        gui.sendText("About to install new agent");
        ResidentAgent newAgent = (ResidentAgent)
```

```
        classObject.newInstance();
      newAgent.install(this);
      gui.sendText("New agent installed");
      agentInstances.add(newAgent);
      numberOfSchedulersPresent++;
    }catch(Exception e){
      gui.sendText("ResidentAgent bad: "+e.toString());
      badAgent = true;
    }
  }
  if(!badAgent){
    gui.sendText("Agent "+
    "numberOfInstancesNeeded==0? "not instantiated.":"instantiated "+
      (numberOfInstancesNeeded==1? "once.": numberOfInstancesNeeded+"
    times")));

    boolean agentAlreadyInstalled = false;
    synchronized(agents){
      agentAlreadyInstalled=agents.containsKey(className);
    }
    if(agentAlreadyInstalled){
      gui.sendText(className+" already resident");
      Vector previousAgents = (Vector)agents.remove(className);
      gui.sendText("Uninstalling previous agent: ");
      gui.sendText("Removing from channels...");
      for(int i=0;i<previousAgents.size();i++){
        ResidentAgent previousAgent = (ResidentAgent)
          previousAgents.get(i);
        if(previousAgent instanceof SaamTalker){
          removeTalkerFromAllChannels(
              (SaamTalker)previousAgent);
        }
        //every ResidentAgent is a SaamListener
        removeListenerFromAllChannels(
            previousAgent);
        ResidentAgent replacement = (ResidentAgent)
          agentInstances.get(i);
        if(previousAgent!=null){
          gui.sendText("Previous agent uninstalling");
          previousAgent.transferState(replacement);
          previousAgent.uninstall();
          previousAgent = null;
        }else{
          gui.sendText("No previous agent installed");
        }
      }
    }
    for (int i=0;i<numberOfInstancesNeeded;i++){
      ResidentAgent replacement = (ResidentAgent)
        agentInstances.get(i);
      notifyAgentCustomers(replacement, className);
      gui.sendText("Notifying customers, agent: "+replacement);
    }
    agents.put(className,agentInstances);
  }
```

```
  }

  /**
   * Here, the ControlExecutive iterates through the Vector of
   * ResidentAgentCustomers and calls the replaceAgent method of
   * each customer, passing the new agent.
   */
  private void notifyAgentCustomers(ResidentAgent ra,
                                          String className){
      Vector customersOfThisAgent = (Vector)
        agentCustomers.get(className);
      if(customersOfThisAgent!=null){
        for(int i=0;i<customersOfThisAgent.size();i++){
          ((ResidentAgentCustomer)
            customersOfThisAgent.get(i)).
              replaceAgent(ra);
        }
        gui.sendText("Agent replaced: "+ra);
        gui.sendText("Customers: "+customersOfThisAgent);

      }
  }

  /**
   * In this method we would reflect into the Class Object and perform
   * a series of policy-related checks to determine whether or not the
   * agent is safe to instantiate.
   */
  private boolean examineAgent(Class classObject){
    //future work..
    return true;
  }

/**
 * This method is called by the Channels this Object has registered to
 * monitor when a talker sends events on those Channels.
 * @param se The SaamEvent to be communicated.
 */
 public synchronized void receiveEvent(SaamEvent se){

 //the ControlExecutive only listens on the Channel between itself and
 //the PacketFactory. Two types of traffic are sent on this Channel,
 //ResidentAgentEvents and MessageEvents

  if(se instanceof ResidentAgentEvent){
    Class classObject = ((ResidentAgentEvent)se).
                                  getClassObject();
    String className = classObject.getName();
    gui.sendText("received residentagent : "+className);
    //16 Jan 2000 akkoc added to determine the serveragent installed


if(className.equals("saam.residentagent.server.ServerAgentSymetric")){
        isServer = true;
      }
```

```
      if (examineAgent(classObject)){
        replaceOldAgent(classObject, className);
      }else{
        //notify someone that the agent failed the inspection
      }
    }else if (se instanceof MessageEvent){

      MessageEvent me = (MessageEvent)se;
      String name =  me.getMessage().getClass().getName();
      gui.sendText("\nreceived messageevent : "+name);
     // System.out.println("received messageevent : "+name);

      //call the appropriate MessageProcessor to handle this Message
      MessageProcessor mp = null;
      mp = (MessageProcessor) messageProcessors.get(name);

      try{
        gui.sendText(" Calling Processor: "+mp.getClass().toString());
              mp.processMessage(me.getMessage());
      }catch(NullPointerException npe){
        //notify the sender that we do not have a
        //processor that is capable of processing this
        //Message.
        gui.sendText(" No Processor Available for " + name);
      }//try-catch
    }//not a ResidentAgentEvent or a MessageEvent
      mainGui.updateDisplay();

  }//receiveEvent()

/**
* Returns the Vector of Interfaces that have been instantiated by this
* ControlExecutive.
* @return The Vector of Interfaces that have been instantiated by this
* ControlExecutive.
*/
public Vector getInterfaces(){
    return interfaces;
}//getInterfaces

  /**
   * The order in which Interfaces are instantiated is preserved.  Here
   * an Object can retrieve a specific Interface by instance number.
   * @param interfaceNumber The instance number of the Interface to be
   * retrieved.
   * @return The nth instance of Interface where n = interfaceNumber.
   */
  public Interface getInterface(int interfaceNumber){
      return (Interface)interfaces.get(interfaceNumber);
  }

  /**
   * Returns the Vector of InterfaceIDs assigned to the Interfaces
   * instantiated by this ControlExecutive.
```

```
   * @return The Vector of InterfaceIDs assigned to the Interfaces
   * instantiated by this ControlExecutive.
   */
  public Vector getInterfaceIDs(){
      return interfaceIDs;
  }//getInterfaces

  /**
   * Returns the Enumeration of Channels that have been instantiated
   * by this ControlExecutive.
   * @return The Enumeration of Channels that have been instantiated
   * by this ControlExecutive.
   */
  public Enumeration getActiveChannels(){
    return activeChannels.elements();
  }//getActiveChannels()

/**
* Returns true if the Channel has been instantiated by this ContExec.
* @return True if the Channel has been instantiated by this ContExec.
*/
public boolean isActiveChannel(int channel_ID){
    return activeChannels.containsKey(new Integer(channel_ID));
}

  /**
   * To determine whether or not a talker is allowed to talk.  If
   * this method returns false, the talker will not be able to talk
   * on any Channels.
   * @param talker The SaamTalker to be verified.
   * @return True if the SaamTalker is allowed to talk.
   */
  private boolean verifyTalker(SaamTalker talker){
    return true;
  }//verifyRequestor()

/**
* To determine whether or not a listener has access to a given Channel.
* This method would be used to implement policy issues related to
access
* control.
* @param listener The listener to be verified.
* @param channel_ID The ID of the Channel.
*/
private boolean verifyChannelAccess(
    SaamListener pl, int channel_ID){
    //here, we would set the policy for channel_ID access.
    //i.e. we can restrict access of certain channel_ID to a
    //select list of listeners.  maybe a Hashtable called
    //"authorizationTable" which contains a Vector of
    //"authorizedListeners" and is keyed on channel_ID.
    return true;
}//verifyAccess()

/**
```

```
* To determine whether or not a talker has access to a given Channel.
* This method would be used to implement policy issues related to
access
* control.
* @param talker The talker to be verified.
* @param channel_ID The ID of the Channel.
*/
private boolean verifyChannelAccess(
    SaamTalker talker, int channel_ID){
    //here, we would set the policy for channel_ID access.
    //i.e. we can restrict access of certain channel_ID to a
    //select list of listeners.  maybe a Hashtable called
    //"authorizationTable" which contains a Vector of
    //"authorizedListeners" and is keyed on channel_ID.
    return true;
}//verifyAccess()

/**
 * As the name implies, this method removes talker from the talker
 * Vectors of all Channels it has registered to talk on.
 * @param talker The talker to be removed.
 */
  public void removeTalkerFromAllChannels(SaamTalker talker){
    if(channelsTalkerHas.containsKey(talker)){
      Vector channels = null;
      synchronized(channelsTalkerHas){
        channels =(Vector)channelsTalkerHas.get(talker);
      }
      Enumeration e = ((Vector)channelsTalkerHas.get(talker)).
        elements();
      while(e.hasMoreElements()){
        Channel thisChannel = (Channel)e.nextElement();
        thisChannel.removeTalker(talker);
        gui.sendText(talker.toString()+
          " removed from channel "+
          thisChannel.getChannel_ID());
        gui.sendText("The Vector: "+channels.toString());
      }
      channelsTalkerHas.remove(talker);
    }
  }

 /**
  * As the name implies, this method removes listener from the listener
  * Vectors of all Channels it has registered to listen on.
  * @param listener The listener to be removed.
  */
  public void removeListenerFromAllChannels(
    SaamListener listener){

    if(channelsListenerHas.containsKey(listener)){
      Vector channels = null;
      synchronized(channelsListenerHas){
        channels = (Vector)channelsListenerHas.get(listener);
      }
```

```
      Enumeration e = channels.elements();
      while(e.hasMoreElements()){
        Channel thisChannel = (Channel)e.nextElement();
        thisChannel.removeListener(listener);
        gui.sendText(listener.toString()+
          " removed from channel "+
          thisChannel.getChannel_ID());
        gui.sendText("The Vector: "+channels.toString());
      }
      channelsListenerHas.remove(listener);
    }
  }

  /**
  *SaamTalkers use this method to attach themselves to a Channel.If this
  * method succeeds, talker will be allowed to transmit events on this
  * Channel.
  * @param talker The talker requesting permission to talk on a Channel.
  * @param channel_ID The ID of the channel to be utilized.
  */
  public void addTalkerToChannel(SaamTalker talker, int channel_ID)
     throws ChannelException {

      if(!verifyTalker(talker)){
        throw new ChannelException("Talking Denied");
      }

      //now test to see whether this channel_ID is within the
      //range of channel_IDs on which this requestor is authorized
      //to talk (policy issue).
      if (!verifyChannelAccess(talker, channel_ID)){
          throw new ChannelException("Access Denied");
      }
      Channel channel = null;
      synchronized(activeChannels){
        channel = (Channel)activeChannels.get(new Integer(channel_ID));
      }
      if(channel==null) {
        channel = new Channel(channel_ID,talker);
      }
      activeChannels.put(new Integer(channel_ID),channel);
        channel.addTalker(talker);
        mainGui.updateDisplay();
      if(channelsTalkerHas.containsKey(talker)){
        synchronized(channelsTalkerHas){
          ((Vector)channelsTalkerHas.get(talker)).
            add(channel);
        }
      }else{
        Vector vectorOfChannels = new Vector();
        vectorOfChannels.add(channel);
        channelsTalkerHas.put(talker, vectorOfChannels);
      }
}//addTalkerToChannel()
```

```
  /**
   * Allows a SaamListener to monitor an emulated UDP port
   * @param listener The listener requesting to monitor a port.
   * @param port The port to be monitored.
   */
  public void monitorPort(SaamListener listener, int port)
    throws PortAccessDeniedException{
    //presumably, the listener has already been verified
    //by the Control Executive and placed on an access
    //list within the EventController.  There is no such
    //access list at this time.
      try{
        if(!hasListener(port)){
         addTalkerToChannel(transportInterface,port);
         addListenerToChannel(listener, port);
//         gui.sendText(listener.toString()+
//           " listening to port: "+port);
        }else {
          gui.sendText(listener.toString()+
            " denied access to port: "+port);
          throw new PortAccessDeniedException("Port in use");
        }
      }catch(ChannelException ce){
        throw new PortAccessDeniedException("Not authorized");
      }//try-catch
  }

/**
*SaamListeners use this method to attach themselves to Channel.If this
* method succeeds, listener will receive all events that are sent on
this
* Channel.
* @param listener The listener requesting to monitor a Channel.
* @param channel_ID The ID of the channel to be monitored.
*/
public void addListenerToChannel(
   SaamListener listener, int channel_ID)
   throws ChannelException{

    if (verifyChannelAccess(listener,channel_ID)){

      Channel channel = null;
      synchronized(activeChannels){
        channel = (Channel)activeChannels.get(new Integer(channel_ID));
      }
      if(channel==null) {
        channel = new Channel(channel_ID,listener);
      }

      //no effect if the Channel is already on the active list
      activeChannels.put(new Integer(channel_ID),channel);
      channel.addListener(listener);
if(channelsListenerHas.containsKey(listener)){
        synchronized(channelsListenerHas){
          ((Vector)channelsListenerHas.get(listener)).
```

```
            add(channel);
        }
      }else{
        Vector vectorOfChannels = new Vector();
        vectorOfChannels.add(channel);
        channelsListenerHas.put(listener, vectorOfChannels);
      }

    }//if
  }//addListenerToChannel()

/**
* Used to determine if any Objects are registered to listen on Channel
* with channel_ID.
* @param channel_ID The ID of the Channel to be queried.
*/
public boolean hasListener(int channel_ID){
    try{
      Channel channel = null;
      synchronized(activeChannels){
        channel = (Channel)activeChannels.get(new Integer(channel_ID));
      }
      return channel.hasListeners();
    }catch(NullPointerException npe){}
      return (false);

}//hasListener()

/**
* Used to determine if any Objects are registered to talk on the
Channel
* with channel_ID.
* @param channel_ID The ID of the Channel to be queried.
*/
public boolean hasTalker(int channel_ID){
    try{
      Channel channel = null;
      synchronized(activeChannels){
        channel = (Channel)activeChannels.get(new Integer(channel_ID));
      }
      return channel.hasTalkers();
    }catch(NullPointerException npe){}
      return (false);

}//hasListener()

/**
 * Once approved to communicate on a channel, a
 * SaamTalker calls this method to actually broadcast
 * events on the channel.  A ChannelException will be
 * thrown if the talker is not registered to talk on
 * the channel contained in the SaamEvent.
 */
 public void talk(SaamEvent event) throws
    ChannelException{
```

```
      SaamTalker talker = event.getTalker();
      int channel_ID = event.getChannel_ID();
      Channel channel = null;
//    synchronized(activeChannels){
      synchronized(theLock) {
// Questions to Dean:
// (1) Is the above sufficient?
// (2) Does this support talking to multiple channels?
// (3) Why are more and more ">>> Ready to ..." msgs printed out?

      channel = (Channel)
      activeChannels.get(new Integer(event.getChannel_ID()));
     // }// old LOCK ENDS HERE
      if(channel.isRegistered(talker)){
        //order the channel to notify its listeners
        gui.sendText(talker.toString()+">>> is Ready to channel.talk");
        channel.talk(event);
        channel.setTimeLastUsed();
      }else{
        gui.sendText("ACCESS DENIED! Unregistered talker: "+
                        talker.toString() +"\n"+
                        "Attempted to talk on channel "+channel_ID);
        throw new ChannelException(
           talker.toString()+" not Registered on "+
           "channel "+channel_ID+".");
       }
      }// new LOCK ens here
  }//talk()

/**
* Displays the status of all Channels that have been instantiated by
* ContExecutive.  Channels are displayed in the ControlExecutive's gui.
* @param msg The text to appear before the channels are displayed.
*/
public void displayActiveChannels(String msg){

/*      gui.sendText("\n"+msg);
      gui.sendText("Active channels:");
      Enumeration e = activeChannels.keys();
      while(e.hasMoreElements()){
        Integer key = (Integer)(e.nextElement());
        Channel channel =
          (Channel)activeChannels.get(key);
        gui.sendText(channel.toString());
      }//while(e.hasMoreElements())
*/
  }//displayActiveChannels()

/**
* Removes a SaamListener from the Vector of listeners associated with
* Channel containing channel_ID
* @param sl The SaamListener to be removed.
* @param channel_ID The ID of the desired Channel.
*/
  public void removeListenerFromChannel(
```

```
    SaamListener sl, int channel_ID){

      Channel channel = null;
      synchronized(activeChannels){
         channel = (Channel)
          activeChannels.get(new Integer(channel_ID));
      }
      channel.removeListener(sl);
  }//closeChannelConnection()

  /**
   * Returns a <code>String</code> representation of this object
   * @return The <code>String</code> representation of this object
   */
  public String toString(){
    return ("Control Executive");
  }

} //end of class CONTROL EXECUTIVE
```

```
//-----------------------------------------
// Filename : PacketFactory.java
// Feb 2000[akkoc/xie]    - modified
// 01Aug99 [Vrable]  - Created
// Project  : SAAM
//-----------------------------------------------------------------------

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;
import java.util.TooManyListenersException;
import java.util.StringTokenizer;
import java.lang.reflect.Constructor;

import saam.net.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;
import saam.residentagent.*;

/**
* A PacketFactory can be used to build SaamPackets for sending or
* to receive SaamPackets and extract their atomic elements.  These
* atomic elements are currently one of two types:  A subclass of
* saam.residentagent.ResidentAgent or a subclass of
* saam.message.Message.<p>
* A sender would instantiate a PacketFactory to build
* Saam Packets.  The PacketFactory's append methods receive
* Message Objects, ResidentAgent Objects, or a String that represents
* the class name of a ResidentAgent as parameters and then dynamically
* construct the appropriate header based on the number of elements
* received and the current time.  The getBytes method is used to
* retrieve the byte array that represents the SAAMPacket that has been
* constructed by this PacketFactory.<p>
* The ControlExecutive uses the PacketFactory to receive and parse
* SaamPackets.
*/
public class PacketFactory extends Thread
  implements SaamTalker, SaamListener{

  private final boolean guiActive = true;
  private SAAMRouterGui gui;
  private ControlExecutive controlExec;
  private boolean started = false;
  private boolean firstEvent = true;
  private boolean bytesRetrieved;
  private byte[] packet,DCMpacket,PNpacket,UCMpacket;
  private byte numberOfMessages;
  private Loader loader;
  private Class message;
  private SaamEvent currentEvent;
  private Thread owner;
```

```
private static int instanceNumber;
private Object theLock = new Object();

/**
 * Use the no-args constructor to begin constructing packets
 * on the sending side.
 */
//no-args constructor doesn't come for free when we have
//another constructor
public PacketFactory(){
  instanceNumber++;
  gui = new SAAMRouterGui(toString() + "("+ instanceNumber+")");
  gui.setTextField("I construct outbound packets");
}

/**
 * This constructor is not available to Objects outside the
 * saam.control package.  The ControlExecutive uses this constructor
 * to receive and parse SAAMPackets.  The PacketFactory passes the
 * atomic elements (either ResidentAgents or Messages) up to the
 * ControlExecutive for further processing.
 * @param controlExec The ControlExecutive that is to receive
 * updates from this PacketFactory.
 */
PacketFactory(ControlExecutive controlExec){
  this();
  gui.setTextField("I Listen for inbound packets");
  this.controlExec=controlExec;
  loader = new Loader();

  //******************************
  //**Listen to desired Channels**
  //******************************
  int channel_ID =
    ProtocolStackEvent.PACKETFACTORY_CHANNEL;
  try{
    controlExec.addListenerToChannel(this, channel_ID);
    gui.sendText("Listening to channel: "+channel_ID);
  }catch(ChannelException ce){
    gui.sendText(ce.toString());
  }//try-catch

  //****************************************
  //**Register to talk on desired Channels**
  //****************************************
  channel_ID = ControlExecutive.SAAM_CONTROL_PORT;
  try{
     controlExec.addTalkerToChannel(this,
       channel_ID);
     gui.sendText("Talking enabled on channel: " + channel_ID);
  }catch(ChannelException ce){
   gui.sendText(ce.toString());
  }
}
```

```
  /**
   * When instantiated to receive packets, the PacketFactory
   * Thread waits until a SAAMPacket arrives, then it calls
   * the processPacket method.
   */
  public void run(){
     while(true){
        try{
          if(!started){
/*          synchronized(this){
              gui.sendText("Waiting...");
              while(!started) wait();
              started=true;
            }
*/
            synchronized(theLock){
              gui.sendText("Waiting...");
              while(!started) theLock.wait();
              started=true;
            }
          }
        }catch(InterruptedException ie){
          gui.sendText(ie.toString());
        }
        gui.sendText("Resumed");
        processPacket();
     }//while(started)
  }

/**
* This method is called by the Channels this Object has registered to
* monitor when a talker sends events on those Channels.
* @param se The SaamEvent to be communicated.

public synchronized void receiveEvent(SaamEvent se){
*/
  public  void receiveEvent(SaamEvent se){

    gui.sendText("\nGot a packet");
    currentEvent=se;
    //check to see if the currentThread has an owner, if it
    //does, notify the owner that the event has arrived.
    //otherwise, just process the packet.
    if(!firstEvent){
      synchronized(theLock){
        theLock.notify();
      }
      if(!started){

        synchronized(theLock){
          started=true;
          theLock.notify();
        }

      }else{
```

```
      processPacket();
    }
  }else{
    firstEvent=false;
    started=true;
    start();
  }
  se=null;
}
/**
 * This method is used to extract the individual Class
 * Objects that are represented in the packet.  These Class
 * Objects are either of type 0 (ResidentAgent) or 1 (Message).<p>
 * If a ResidentAgent is received, a Class Object is created
 * that represents the agent.  That Class Object is then sent to
 * the ControlExecutive for screening and agent instantiation.<p>
 * If a Message is received, that Message is instantiated and sent
 * to the ControlExecutive for further processing.
 */

private void processPacket() {
  int channel = currentEvent.getChannel_ID();
  String eventSource = (String)currentEvent.getSource();

  //packet is a byte array
  packet = ((ProtocolStackEvent)currentEvent).getPacket();


  //see saam.util for PrimitiveConversions and Array classes
  long timeStamp = PrimitiveConversions.getLong(
      Array.getSubArray(packet,0,8));
  numberOfMessages=packet[8];
  gui.sendText("packet arrived: " +
    "\n  source:        " + eventSource +
    "\n  channel:       " + channel +
    "\n  size:          " + packet.length +
    "\n  # of Messages: " + numberOfMessages +
    "\n  timeStamp:     " + timeStamp);

  //now we trim the packet by removing the header.
  packet = Array.getSubArray(packet,9,packet.length);

  //used to track the current position in the array.
  int index = 0;

          for(int i=1;i<=numberOfMessages;i++){
    gui.sendText("\nProcessing Element["+i+"]:");
  //      int index = 0;
            byte type = packet[index++];

    switch(type){
      case 0:
      case 1:
    //extract and process each atomic element of the packet
        //separately.  Here we assume the packet is a properly
```

```
//formatted SAAMPacket when it arrives, and that the
//length is less than the max allowed.

gui.sendText("  type:   "+type);
//retrieve the number of bytes the class name occupies
byte nameLength = packet[index++];

//extract the name of the class file as a byte array
 byte[] elementNameArray = Array.getSubArray(
 packet,index, index+nameLength);
 index+=nameLength;

 //convert the name back into a String
 String elementName = new String(elementNameArray);
        gui.sendText("  Name: "+elementName);

 //retrieve the length of the Object
        short length = PrimitiveConversions.getShort(
  Array.getSubArray(packet,index,index+2));
  gui.sendText("  Length:       "+length);
  index+=2;

  //retrieve the bytecode of the Object
  byte[] bytes = Array.getSubArray(
  packet,index,index+length);
  index+=length;

if (type == 0){
gui.sendText("This is a ResidentAgent");
//Assume this class is of type ResidentAgent
try{
//Attempt to define the class using the current
//class loader.
loader.defClass(elementName, bytes);
}catch(LinkageError le){
//If the loader already has a definition for the class
//a LinkageError will be thrown.  If this happens, we
//need to instantiate a new class loader and use it to
//define the class.  A nice little trick we learned from
//page 55 of Jason Hunter's "Java Servlet Programming" book.
gui.sendText(le.toString());
gui.sendText("Class was previously loaded...");
gui.sendText("Replacing old ClassLoader...");
Loader newLoader = new Loader();
   newLoader.defClass(elementName, bytes);
}
      try{
        //message is of type Class.
        message = Class.forName(elementName, true, loader);
      }catch(ClassNotFoundException cnfe){
        gui.sendText(cnfe.toString());
      }
      gui.sendText(message.toString());
      ResidentAgentEvent rae = new ResidentAgentEvent(
        eventSource,
```

```
        this,
        ControlExecutive.SAAM_CONTROL_PORT,
        message);
    try{
       gui.sendText("Forwarding on channel "+
           ControlExecutive.SAAM_CONTROL_PORT);
       controlExec.talk(rae);
    }catch(ChannelException tde){
        gui.sendText(tde.toString());
    }
                    }else {
    gui.sendText("This is a Message");
       //Assume this class is of type Message.
    try{
     //message is of type Class.
     message = Class.forName(elementName);
       }catch(ClassNotFoundException cnfe){
   {gui.sendText("Bytecode for: "+elementName+
           " not found.");
    }
    }

   try{
   //Call the constructor from within this Class that
   //takes a byte array as its only argument
   Constructor cons = message.getConstructor(
           new Class[] {byte[].class});

   //Create the instance of this Message
   Message instance = (Message)cons.newInstance(
             new Object[] {bytes});
   gui.sendText(instance.toString());
   MessageEvent me = new MessageEvent(eventSource, this,
           ControlExecutive.SAAM_CONTROL_PORT,  instance);
   //send this MessageEvent on the Control port.
   try{
      gui.sendText("Forwarding on channel "+
           ControlExecutive.SAAM_CONTROL_PORT);
           controlExec.talk(me);
   }catch(ChannelException tde){
       gui.sendText("problem occured here ");
       gui.sendText(tde.toString());
   }
   }catch(Exception e){
   //need to notify sender that we have no classfile
   //with this name
   gui.sendText(e.toString());
   }//try-catch
                    }
break;

case 4:
  gui.sendText("This is a DCM  Message");

  try{
```

```
  DCM  dcm = new DCM(packet);
  gui.sendText(dcm.toString());
  MessageEvent me = new MessageEvent( eventSource, this,
                    ControlExecutive.SAAM_CONTROL_PORT,dcm);
  //send this MessageEvent on the Control port.
  try{
      gui.sendText("Forwarding on channel "+
         ControlExecutive.SAAM_CONTROL_PORT);
         controlExec.talk(me);
  }catch(ChannelException tde){
      gui.sendText(tde.toString());
    }
  }catch(Exception e){
    gui.sendText(e.toString());
}//try-catch
break;

case 5:
  gui.sendText("This is a UCM  Message");
  try{
    UCM ucm  =  new UCM(packet);
    gui.sendText("\n"+ucm.toString());
    MessageEvent me = new MessageEvent( eventSource, this,
          ControlExecutive.SAAM_CONTROL_PORT,ucm);
    //send this MessageEvent on the Control port.
    try{
      gui.sendText("Forwarding on channel "+
         ControlExecutive.SAAM_CONTROL_PORT);
      controlExec.talk(me);
    }catch(ChannelException tde){
      gui.sendText(tde.toString());
    }
  }catch(Exception e){
    gui.sendText(e.toString());
}//try-catch
break;

case 6:
  gui.sendText("This is a ParentNotification Message");
  //Assume this class is of type Message.
  try{
    ParentNotification pn = new ParentNotification(packet);
    MessageEvent me = new MessageEvent( eventSource, this,
                      ControlExecutive.SAAM_CONTROL_PORT,pn);
    //send this MessageEvent on the Control port.
    try{
      gui.sendText("Forwarding on channel "+
      ControlExecutive.SAAM_CONTROL_PORT);
      controlExec.talk(me);
    }catch(ChannelException tde){
      gui.sendText(tde.toString());
    }
  }catch(Exception e){
    gui.sendText(e.toString());
}//try-catch
```

```
        break;

        default:
          gui.sendText("Packet type unrecognized: "+type);
          //packet type is unrecognized.  Here we could
          //extract a channel_ID that could be embedded
          //in the packet, and then send the unrecognized
          //element on that channel.
    }//end switch
  }//for
  started=false;
}//processPacket()

/**
 * This method can be used to append a Message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getBytes method.
 * @param me The Message to be appended.
 */
public void append(Message me){
  if(bytesRetrieved){
    packet=null;
    numberOfMessages=0;
    bytesRetrieved = false;
  }
  byte type = me.getType();


    String name = me.getClass().getName();
    gui.sendText("appending "+name);
    byte nameLength = (byte)name.getBytes().length;
    byte[] parameters = me.getBytes();

  //here we could check the length of the parameter array supplied
  //with the length returned from the length() method call.
    short paramLength = (short)parameters.length;
    //now append the Message to the packet byte array
    packet = Array.concat(packet,type);
    packet = Array.concat(packet,nameLength);
    packet = Array.concat(packet,name.getBytes());
    packet = Array.concat(packet,
             PrimitiveConversions.getBytes(paramLength));
    packet = Array.concat(packet,parameters);
    //increment the count of messages in this packet
     numberOfMessages++;

    gui.sendText("Appended Message:" +
    "\n  Type:           " + type +
    "\n  name:           " + name +
    "\n  param length:   " + paramLength +
    "\n  # of elements:  " + numberOfMessages +
    "\n  packet length:  " + packet.length+"\n");
```

```
}  //end of append

 /**
 * This method can be used to append a DCM message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getDCMBytes method.
 * @param downWard The DCM message to be appended.
 */

public void appendDCM( DCM downWard){
  gui.sendText(" Appends a dcm before sending downward with lengh");
  DCMpacket = Array.concat(DCMpacket,downWard.getBytes());
}//end of appendDCm

/**
 * This method can be used to append a PN message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getPNBytes method.
 * @param downWard The PN message to be appended.
 */
public void appendPN( ParentNotification pn){
  gui.sendText(" Appending a PN before sending downward with lengh");
  PNpacket = Array.concat(PNpacket,pn.getBytes());
  gui.sendText("after appending PN is "+PNpacket.length);

}//end of appendDCm

/**
 * This method can be used to append a PN message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getPNBytes method.
 * @param downWard The PN message to be appended.
 */
public void appendUCM( UCM upWard){
  gui.sendText(" Appending a UCM message before sending upward");
  UCMpacket = Array.concat(packet,upWard.getBytes());
}//end of appendUCM

/**
 * This method can be used to append a ResidentAgent to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getBytes method.
 * @param ra The ResidentAgent to be appended.
 */
public void append(ResidentAgent ra) throws IOException{
  String name = ra.getClass().getName();
  append(name);
}
/**
 * This method can be used to append a ResidentAgent by name to an
 * outgoing SAAMPacket.  To later retrieve the entire packet
 * (with header) as a byte array, call the getBytes method.
 * @param residentAgentClassName The String name of the ResidentAgent
 * classfile to be appended.
```

```
   */
  public void append(String residentAgentClassName)
    throws IOException{
    if(bytesRetrieved){
      packet=null;
      numberOfMessages=0;
      bytesRetrieved = false;
    }
    byte type = 0;
    String name = residentAgentClassName;
//  String fileName =
"C:\\WINNT\\Profiles\\administrator\\Desktop\\Java\\saamjuly\\saamxpand
1"+File.separatorChar +
    String fileName = "C:\\hakkoc"+File.separatorChar +
      residentAgentClassName.replace('.',File.separatorChar);
    fileName+=".class";
    gui.sendText("File name: "+fileName);
    FileInputStream fis = null;
    try{
      fis = new FileInputStream(fileName);
    }catch(IOException ioe){
      throw new IOException(
        "Problem reading ResidentAgent: "+fileName);
    }
    byte nameLength = (byte)name.getBytes().length;
    byte[] byteCode = new byte[fis.available()];
    short length = (short)fis.read(byteCode);

    packet = Array.concat(packet,type);
    packet = Array.concat(packet,nameLength);
    packet = Array.concat(packet,name.getBytes());
    packet = Array.concat(packet,
      PrimitiveConversions.getBytes(length));
    packet = Array.concat(packet,byteCode);
    numberOfMessages++;

    gui.sendText("Appended ResidentAgent:" +
      "\n  Type:              " + type +
      "\n  name:              " + name +
      "\n  byteCode length:   " + length +
      "\n  # of elements:     " + numberOfMessages +
      "\n  packet length:     " + packet.length+"\n");

  }

  /**
   * Appends a header to the byte array.  The header conforms
   * to the structure of a SAAMHeader.
   */
  private void appendHeader(){
    byte[] timeStamp = PrimitiveConversions.getBytes(
      System.currentTimeMillis());
      packet = Array.concat(numberOfMessages,packet);
      packet = Array.concat(timeStamp,packet);
      gui.sendText("Appended header:"+
```

```
        "\n  timeStamp:    "+PrimitiveConversions.getLong(
          Array.getSubArray(packet,0,8))+
        "\n  # of updates: "+packet[8] +
        "\n  packet length: "+packet.length+"\n");

  }

  /**
   * Returns a byte array that conforms to the structure of
   * a SAAMPacket.
   * @return A byte array that conforms to the structure of
   * a SAAMPacket.
   */
  public byte[] getBytes(){
    appendHeader();
    bytesRetrieved = true;
    return packet;
  }

 /**
 * Returns a byte array that conforms to the structure of a DCMPacket.
 * @return A byte array that conforms to the structure of DCMPacket.
 */
  public byte[] getDCMBytes(){
    return DCMpacket;
  }

  /**
   * Returns a byte array that conforms to the structure of a PNPacket.
   * @return A byte array that conforms to the structure of PNPacket.
   */
  public byte[] getPNBytes(){
    return PNpacket;
  }

/**
* Returns a byte array that conforms to the structure of a UCMPacket.
* @return A byte array that conforms to the structure of UCMPacket.
*/
public byte[] getUCMBytes(){
  return UCMpacket;
}

/**
 * Returns the current length of the packet.
 * @return The current length of the packet.
 */
 public int length(){
    try{
      return packet.length;
    }catch(NullPointerException npe){
      return 0;
    }
  }
```

```
  /**
   * Returns a <code>String</code> representation of this object
   * @return The <code>String</code> representation of this object
   */
  public String toString(){
    return "Packet Factory";
  }

}
```

```
//-----------------------------------------
// Filename : Demo_2_ServerS_4_RouterS.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project   : SAAM
//-----------------------------------------------------------------------

package saam.demo;

import saam.*;
import saam.control.*;
import saam.message.*;
import saam.residentagent.*;
import saam.router.*;
import saam.net.*;
import saam.util.*;
import java.net.*;
import java.io.*;
import java.util.Vector;
import java.util.Enumeration;

public class Demo_2_ServerS_4_RouterS{
   private PacketFactory packet = new PacketFactory();

   private InetAddress
           destMain,destBackUp,destA,destB,destC,destD,destE;
   private int destEmulationPort=9010;    //SAAM UDP emulation port
   private DemoGui gui = new DemoGui("DemoStation_2_Server_4_Routers");

   Configuration cfMain = null;  //FOR MAIN SERVER
   Configuration cfBackUp = null; //For BackUp Server

   //Different values may be sent to each server and router
   private int timeScaleForMain = 30;
   private int timeScaleForBackUp = 30;
   private int timeScaleForRouter_A = 550;
   private int timeScaleForRouter_B = 400;
   private int timeScaleForRouter_C = 550;
   private int timeScaleForRouter_D = 400;
  // private int timeScaleForRouter_E = 400;

   private static final byte MAIN_SERVER_TYPE_ID = 0;
   private static final byte  BACK_UP_SERVER_TYPE_ID = 1;

   private static final int MAIN_SERVER_FLOW_ID = 1;
   private static final int BACK_UP_SERVER_FLOW_ID = 3;

   private static final byte METRIC_TYPE = 0;
  //Metric Type 0->For Symmetric (first arriving best), 1->For Hopcount

   private static final int MAIN_REFRESH_CYCLE_TIME = 2000;// In msec.
   private static final int BACK_UP_REFRESH_CYCLE_TIME = 2000;//Inmsec.

   private static final int MAIN_GLOBALTIME_TO_WAIT = 200;// In msec.
   private static final int BACK_UP_GLOBALTIME_TO_WAIT = 200;//In msec.
```

```
private String[] coreAgents =
   {"saam.residentagent.router.Scheduler",
    "saam.residentagent.router.ARPCache",
    "saam.residentagent.router.FlowRoutingTable"};

public static void main(String args[]){
    Demo_2_ServerS_4_RouterS test = new  Demo_2_ServerS_4_RouterS();
     System.exit(0);
}

public  Demo_2_ServerS_4_RouterS(){
   try{
       gui.setTextField("My IP: "+
       InetAddress.getLocalHost().getHostAddress());
   }catch(UnknownHostException uhe){
     gui.sendText(uhe.toString());
   }

    try{
       destMain= InetAddress.getByName("131.120.8.135"); //server
       destBackUp= InetAddress.getByName("131.120.8.138");//backup
       destA= InetAddress.getByName("131.120.8.147"); //Router A
       destB= InetAddress.getByName("131.120.8.137"); //Router B
       destC= InetAddress.getByName("131.120.9.46"); //Router C
       destD= InetAddress.getByName("131.120.8.155"); //Router D
     //  destE= InetAddress.getByName("127.0.0.1"); //Router E
    }catch(UnknownHostException uhe){
      gui.sendText(uhe.toString());
    }


     //Initilizing interfaces on  MAIN Server
     Vector serverInterfaceM = new Vector();
     Vector serverEmTableM = new Vector();
     Vector serverArpCacheM = new Vector();

     byte  serverMacM = 0;
     byte  serverNextMacM = 1;
     try{
     IPv6Address serIntAdM = new IPv6Address(
        IPv6Address.getByName("99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1")
                                                    .getAddress());
        IPv6Address serNextHopM = new IPv6Address(
        IPv6Address.getByName("99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2")
                                                    .getAddress());
     InetAddress serNextV4M = InetAddress.getByName("131.120.8.147");

     //for demohello message
     serverInterfaceM.add( new InterfaceID( serIntAdM,serverMacM));
     //for EmulationTableEntry message
     serverEmTableM.add(newEmulationTableEntry
                                        (serNextHopM,serNextV4M));
     //for ARPCache
     serverArpCacheM.add( new
```

```
                        ARPCacheEntry(serNextHopM,serverNextMacM));
     cfMain = new Configuration(MAIN_SERVER_TYPE_ID,
             MAIN_SERVER_FLOW_ID,METRIC_TYPE,
             MAIN_REFRESH_CYCLE_TIME*timeScaleForMain,
             MAIN_GLOBALTIME_TO_WAIT*timeScaleForRouter_A );


     }catch(UnknownHostException uhe){
      gui.sendText(uhe.toString());
    }

     //initiating BACKUP Server
     //Initilizing interfaces on Server
     Vector serverInterfaceB = new Vector();
     Vector serverEmTableB = new Vector();
     Vector serverArpCacheB = new Vector();

     byte  serverMacB = 13;
     byte  serverNextMacB = 12;
     try{
     IPv6Address serIntAdB = new IPv6Address(
        IPv6Address.getByName("99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2")
                                                        .getAddress());
        IPv6Address serNextHopB = new IPv6Address(
         IPv6Address.getByName("99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1")
                                                        .getAddress());
     InetAddress serNextV4B = InetAddress.getByName("131.120.8.155");

//for demohello message
serverInterfaceB.add( new InterfaceID( serIntAdB,serverMacB));
//for EmulationTableEntry message
serverEmTableB.add(new EmulationTableEntry(serNextHopB,serNextV4B));
//for ARPCache
serverArpCacheB.add( new ARPCacheEntry(serNextHopB,serverNextMacB));

cfBackUp = new Configuration(BACK_UP_SERVER_TYPE_ID,
                BACK_UP_SERVER_FLOW_ID,METRIC_TYPE,
                BACK_UP_REFRESH_CYCLE_TIME
                *timeScaleForBackUp,BACK_UP_GLOBALTIME_TO_WAIT*
                timeScaleForRouter_A );

}catch(UnknownHostException uhe){
   gui.sendText(uhe.toString());
}


    //ROUTER A
    Vector routerAInterfaces = new Vector();
    Vector routerAEmTable = new Vector();
    Vector routerAArpCache = new Vector();
       //interface-1
    byte routerAMacs_1 = 1;
    byte routerANextMac_1 = 0;
    try{
```

```
IPv6Address routerAInt_1 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2")
                                                .getAddress());
IPv6Address routerANextHop_1 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1")
                                                 .getAddress());
InetAddress routerANextV4_1 = InetAddress.getByName("131.120.8.135");
routerAInterfaces.add( new InterfaceID( routerAInt_1,routerAMacs_1));
routerAEmTable.add(new
EmulationTableEntry(routerANextHop_1,routerANextV4_1));
routerAArpCache.add( new
ARPCacheEntry(routerANextHop_1,routerANextMac_1));
}catch(UnknownHostException uhe){
    gui.sendText(uhe.toString());
}


//interface-2
byte routerAMacs_2 = 2;
byte routerANextMac_2 = 5;
try{
IPv6Address routerAInt_2 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.1")
                                                   .getAddress());
IPv6Address routerANextHop_2 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2")
                                                   .getAddress());
InetAddress routerANextV4_2 = InetAddress.getByName("131.120.8.137");

routerAInterfaces.add( new InterfaceID( routerAInt_2,routerAMacs_2));
routerAEmTable.add(new
EmulationTableEntry(routerANextHop_2,routerANextV4_2));
routerAArpCache.add( new
ARPCacheEntry(routerANextHop_2,routerANextMac_2));
}catch(UnknownHostException uhe){
     gui.sendText(uhe.toString());
}


//interface-3
byte routerAMacs_3 = 3;
byte routerANextMac_3 = 10;
try{
IPv6Address routerAInt_3 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.1")
.getAddress());
IPv6Address routerANextHop_3 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2")
.getAddress());

InetAddress routerANextV4_3 = InetAddress.getByName("131.120.8.155");

routerAInterfaces.add( new InterfaceID( routerAInt_3,routerAMacs_3));
routerAEmTable.add(new
EmulationTableEntry(routerANextHop_3,routerANextV4_3));
```

```
routerAArpCache.add( new
ARPCacheEntry(routerANextHop_3,routerANextMac_3));
}catch(UnknownHostException uhe){
   gui.sendText(uhe.toString());
}
//interface-4
byte routerAMacs_4 = 4;
byte routerANextMac_4 = 7;
try{
IPv6Address routerAInt_4 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1")
.getAddress());
IPv6Address routerANextHop_4 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2")
.getAddress());
       InetAddress routerANextV4_4 =
InetAddress.getByName("131.120.9.46");

routerAInterfaces.add( new InterfaceID( routerAInt_4,routerAMacs_4));
routerAEmTable.add(new
EmulationTableEntry(routerANextHop_4,routerANextV4_4));
routerAArpCache.add( new
ARPCacheEntry(routerANextHop_4,routerANextMac_4));
catch(UnknownHostException uhe){
   gui.sendText(uhe.toString());
}

//ROUTER B

Vector routerBInterfaces = new Vector();
Vector routerBEmTable = new Vector();
Vector routerBArpCache = new Vector();
   //interface-1
byte routerBMacs_1 = 5;
byte routerBNextMac_1 = 2;
try{
   IPv6Address routerBInt_1 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2")
.getAddress());
IPv6Address routerBNextHop_1 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.1")
.getAddress());
 InetAddress routerBNextV4_1 = InetAddress.getByName("131.120.8.147");
 routerBInterfaces.add( new InterfaceID( routerBInt_1,routerBMacs_1));
 routerBEmTable.add(new
       EmulationTableEntry(routerBNextHop_1,routerBNextV4_1));
 routerBArpCache.add( new
ARPCacheEntry(routerBNextHop_1,routerBNextMac_1));
 }catch(UnknownHostException uhe){
    gui.sendText(uhe.toString());
 }
 //interface-2
 byte routerBMacs_2 = 6;
 byte routerBNextMac_2 = 9;
 try{
```

```
IPv6Address routerBInt_2 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1")
.getAddress());
IPv6Address routerBNextHop_2 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.2")
.getAddress());
InetAddress routerBNextV4_2 = InetAddress.getByName("131.120.8.155");

routerBInterfaces.add( new InterfaceID( routerBInt_2,routerBMacs_2));
routerBEmTable.add(new
EmulationTableEntry(routerBNextHop_2,routerBNextV4_2));
routerBArpCache.add( new
ARPCacheEntry(routerBNextHop_2,routerBNextMac_2));

}catch(UnknownHostException uhe){
   gui.sendText(uhe.toString());
}

// ROUTER C
   Vector routerCInterfaces = new Vector();
   Vector routerCEmTable = new Vector();
   Vector routerCArpCache = new Vector();
   //interface-1
   byte routerCMacs_1 = 7;
   byte routerCNextMac_1 = 4;
   try{
  IPv6Address routerCInt_1 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.2").getAddress
());
       IPv6Address routerCNextHop_1 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.3.0.0.0.0.0.0.0.0.0.0.1").getAddress
());
       InetAddress routerCNextV4_1 =
InetAddress.getByName("131.120.8.147");

       routerCInterfaces.add( new InterfaceID(
routerCInt_1,routerCMacs_1));
       routerCEmTable.add(new
EmulationTableEntry(routerCNextHop_1,routerCNextV4_1));
       routerCArpCache.add( new
ARPCacheEntry(routerCNextHop_1,routerCNextMac_1));

        }catch(UnknownHostException uhe){
        gui.sendText(uhe.toString());
      }
           //interface-2
       byte routerCMacs_2 =  8;
       byte routerCNextMac_2 = 11;
       try{
       IPv6Address routerCInt_2 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1").getAddress
());
       IPv6Address routerCNextHop_2 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.2").getAddress
());
```

```
InetAddress routerCNextV4_2 = InetAddress.getByName("131.120.8.155");

routerCInterfaces.add( new InterfaceID( routerCInt_2,routerCMacs_2));
routerCEmTable.add(new
EmulationTableEntry(routerCNextHop_2,routerCNextV4_2));
       routerCArpCache.add( new
ARPCacheEntry(routerCNextHop_2,routerCNextMac_2));
   }catch(UnknownHostException uhe){
      gui.sendText(uhe.toString());
   }

//ROUTER D
Vector routerDInterfaces = new Vector();
Vector routerDEmTable = new Vector();
Vector routerDArpCache = new Vector();
//interface-1
byte routerDMacs_1 = 10;
byte routerDNextMac_1 = 3;
try{
IPv6Address routerDInt_1 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2").getAddress
());
       IPv6Address routerDNextHop_1 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.1").getAddress
());
InetAddress routerDNextV4_1 = InetAddress.getByName("131.120.8.147");

       routerDInterfaces.add( new InterfaceID(
routerDInt_1,routerDMacs_1));
       routerDEmTable.add(new
EmulationTableEntry(routerDNextHop_1,routerDNextV4_1));
       routerDArpCache.add( new
ARPCacheEntry(routerDNextHop_1,routerDNextMac_1));
        }catch(UnknownHostException uhe){
        gui.sendText(uhe.toString());
}
//interface-2
byte routerDMacs_2 = 9;
byte routerDNextMac_2 = 6;
try{
IPv6Address routerDInt_2 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.2").getAddress
());
IPv6Address routerDNextHop_2 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.5.0.0.0.0.0.0.0.0.0.0.1").getAddress
());
InetAddress routerDNextV4_2 = InetAddress.getByName("131.120.8.137");

routerDInterfaces.add( new InterfaceID( routerDInt_2,routerDMacs_2));
       routerDEmTable.add(new
EmulationTableEntry(routerDNextHop_2,routerDNextV4_2));
       routerDArpCache.add( new
ARPCacheEntry(routerDNextHop_2,routerDNextMac_2));
   }catch(UnknownHostException uhe){
     gui.sendText(uhe.toString());
```

```
    }
     //interface-3
     byte routerDMacs_3 = 12;
     byte routerDNextMac_3 = 13;
     try{
     IPv6Address routerDInt_3 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.1").getAddress
());
        IPv6Address routerDNextHop_3 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.6.0.0.0.0.0.0.0.0.0.0.2").getAddress
());
        InetAddress routerDNextV4_3 =
InetAddress.getByName("131.120.8.138");

        routerDInterfaces.add( new InterfaceID(
routerDInt_3,routerDMacs_3));
        routerDEmTable.add(new
EmulationTableEntry(routerDNextHop_3,routerDNextV4_3));
        routerDArpCache.add( new
ARPCacheEntry(routerDNextHop_3,routerDNextMac_3));
         }catch(UnknownHostException uhe){
         gui.sendText(uhe.toString());
       }

    //interface-4
    byte routerDMacs_4 = 11;
    byte routerDNextMac_4 = 8;
    try{
        IPv6Address routerDInt_4 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.2").
getAddress());
IPv6Address routerDNextHop_4 = new IPv6Address(
IPv6Address.getByName("99.99.99.99.4.0.0.0.0.0.0.0.0.0.0.1")
.getAddress());
InetAddress routerDNextV4_4 = InetAddress.getByName("131.120.9.46");

routerDInterfaces.add( new InterfaceID( routerDInt_4,routerDMacs_4));
routerDEmTable.add(new
EmulationTableEntry(routerDNextHop_4,routerDNextV4_4));
routerDArpCache.add( new
ARPCacheEntry(routerDNextHop_4,routerDNextMac_4));
    }catch(UnknownHostException uhe){
      gui.sendText(uhe.toString());
    }



//FIRST STAND UP Main  SERVER!!!!!!
InitServer(serverInterfaceM, serverEmTableM, serverArpCacheM,
destMain,cfMain);
try{
    Thread.sleep(1000);
      }catch(InterruptedException ie){
       gui.sendText("problem afetr initserver therad sleep");
      }
```

```
//Backup
InitServer(serverInterfaceB, serverEmTableB, serverArpCacheB,
destBackUp,cfBackUp);
   try{
      Thread.sleep(1000);
   }catch(InterruptedException ie){
   gui.sendText("problem afetr initserver therad sleep");
   }


//NOW STAND-UP THE ROUTERS!!!!!!!!!

// start Router A
InitRouter( routerAInterfaces, routerAEmTable, routerAArpCache,
                           destA, timeScaleForRouter_A );

     try{
        Thread.sleep(1000);
      }catch(InterruptedException ie){
        gui.sendText("problem afetr initrouter thread sleep");
      }

    // start Router B
      InitRouter( routerBInterfaces, routerBEmTable, routerBArpCache,
                  destB ,  timeScaleForRouter_B );

      try{
        Thread.sleep(1000);
      }catch(InterruptedException ie){
        gui.sendText("problem after initrouter thread sleep");
      }
     // start Router C
   InitRouter( routerCInterfaces, routerCEmTable, routerCArpCache,
                           destC , timeScaleForRouter_C );

      try{
        Thread.sleep(1000);
      }catch(InterruptedException ie){
        gui.sendText("problem afetr initserver thread sleep");
      }

      // start Router D
     InitRouter( routerDInterfaces, routerDEmTable, routerDArpCache,
                        destD,  timeScaleForRouter_D );

      try{
        Thread.sleep(4000);
      }catch(InterruptedException ie){
        gui.sendText("problem afetr initserver thread sleep");
      }

  /*    // start Router E
       InitRouter( routerEInterfaces, routerEEmTable, routerEArpCache,
```

```
                          specialAgents,  moreResAgents, destE,
timeScaleForRouter_E );

      try{
        Thread.sleep(4000);
      }catch(InterruptedException ie){
        gui.sendText("problem afetr initserver thread sleep");
      } */

 } //end DemoStation() constructor

public void InitRouter(Vector routerInterfaces,
                  Vector routerEmTable, Vector routerArpCache,
                  InetAddress dest, int tsForRouter){

//add router InterfaceIDs -- may have to use DemoHello messages instead

      DemoHello helloMessage = new DemoHello(routerInterfaces);
      packet.append(helloMessage);

      try{
        //now append some ResidentAgents...
        //first the agents that are necessary for the
        //protocol stack
        for(int i=0;i<coreAgents.length;i++){
         packet.append(coreAgents[i]);
        }

        //then any additional agents for the specific host

      }catch(IOException ioe){
         gui.sendText(ioe.toString());
         gui.sendText(" problem in initrouter coreagents for block ");
      }

      packet.append(new TimeScale(tsForRouter));
      //add entries to the EmulationTable
      Enumeration e1 = routerEmTable.elements();
      while(e1.hasMoreElements()){
         packet.append( (EmulationTableEntry)( e1.nextElement()));

      } //end of while

      //add entries to the ARPCache
       Enumeration e2 = routerArpCache.elements();
      while(e2.hasMoreElements()){
         packet.append((ARPCacheEntry) e2.nextElement());

      } //end of while

//now send the packet
byte[] packetArray = packet.getBytes();
gui.sendText("#of messages: "+packetArray[8]);//peeks inside packet
try{
         Socket socket = new Socket(dest,destEmulationPort);
```

```
        socket.setTcpNoDelay(true);


gui.sendText("destination= "+dest+" destEmuPort= "+ destEmulationPort);
OutputStream os = socket.getOutputStream();

os.write(packetArray);
// os.flush();
os.close();
socket.close();
}catch(Exception e){
gui.sendText(e.toString());
gui.sendText(" problem in initrouter socket try block ");
}
gui.sendText("Packet sent to "+dest.getHostAddress());
gui.sendText("Length: "+packetArray.length);
}//end InitRouter()

public void InitServer( Vector serverInterface,
              Vector serverEmTable, Vector serverArpCache,
              InetAddress destS,Configuration cf){


      DemoHello helloMessage = new DemoHello(serverInterface);
      packet.append(helloMessage);

      try{
        //now append some ResidentAgents...
        //first the agents that are necessary for the
        //protocol stack
        for(int i=0;i<coreAgents.length;i++){
         packet.append(coreAgents[i]);
        }
        //then any additional agents for the specific host
        packet.append("saam.residentagent.server.ServerAgentSymetric");

      }catch(IOException ioe){
         gui.sendText(ioe.toString());
      }

      //add entries to the Server's EmulationTable
      Enumeration e1 = serverEmTable.elements();
      while(e1.hasMoreElements()){
         packet.append( (EmulationTableEntry)( e1.nextElement()));

      } //end of while

      //add entries to the Server's ARPCache

      Enumeration e2 = serverArpCache.elements();
      while(e2.hasMoreElements()){
         packet.append( (ARPCacheEntry)( e2.nextElement()));

      } //end of while
```

```
    //To send CONFIGURATION INFORMATION
    packet.append( cf );

    //now send the packet
    byte[] packetArray = packet.getBytes();
    gui.sendText("#of messages send to server:"+packetArray[8]);
      try{
        Socket socket = new Socket(destS,destEmulationPort);
        socket.setTcpNoDelay(true);
        gui.sendText("destServer="+destS+"destEmuPort="+
                                    destEmulationPort);
        OutputStream os = socket.getOutputStream();
        os.write(packetArray);
        //   os.flush();

        os.close();
        socket.close();
      }catch(Exception e){
        gui.sendText(e.toString());
        gui.sendText(" problem in initserver socket try block ");
      }
      gui.sendText("Packet sent to "+destS.getHostAddress());
      gui.sendText("Length: "+packetArray.length);
   }//end InitServer()

}//end class DemoStation
```

```
//-----------------------------------------
// Filename : RoutingAlgorithm.java
// Feb 2000[akkoc/xie]   - modified
// 01Aug99 [Vrable]  - Created
// Project  : SAAM
//-----------------------------------------------------------------------

package saam.router;

import java.net.UnknownHostException;
import java.util.*;
import saam.router.Interface;
import saam.router.*;
import saam.*;
import saam.control.*;
import saam.residentagent.*;
import saam.residentagent.router.*;
import saam.util.*;
import saam.event.*;
import saam.net.*;
import saam.message.*;

/**
* The RoutingAlgorithm looks in the inbound queues of all Interfaces on
* this router and extracts packets based on a round-robin
* algorithm.  Dequeued packets are looked into to determine the
destination.
* If the packet is destined for any Interface on this router, the
packet
* is forwarded to the application layer; otherwise the packet is
forwarded
* to the outbound interface.
*/
public class RoutingAlgorithm implements ResidentAgentCustomer,
     SaamTalker, SaamListener, Runnable{

  /**
   * The number of bytes in a SAAM network address.  This
   * variable is used in the routePacket() method to determine
   * the outbound interface when ARPing.
   */
  private static final int bytesToCheck = 5;

  private static final int DATAGRAM_ROUTING = 0;

  /**
   * The agentTypes the RoutingAlgorithm registers to process.
   */
  private static final String[] agentTypes =
    {"saam.residentagent.router.ARPCache",
     "saam.residentagent.router.FlowRoutingTable"};

  private static ResidentAgent arpCache, flowRoutingTable;
```

```
private ControlExecutive controlExec;
private SAAMRouterGui gui = new SAAMRouterGui("RoutingAlgorithm");
private Vector interfaces = new Vector();
private int interfaceCount;
private int populatedQueues;
private Interface outboundInterface;
private boolean started;
private long packetCounter;
private Object theLock= new Object();
private Object populatedQueueLock = new Object();

/**
* The ControlExecutive instantiates the RoutingAlgorithm, passing it
* a copy of the ControlExecutive so the RoutingAlgorithm can call
* methods on that ControlExecutive.  The ControlExecutive instantiates
* the arpCache ResidentAgent and passes it to the RoutingAlgorithm.
* @param controlExec The ControlExecutive this RoutingAlgorithm will
* use to perform certain operations.
* @param arpCache The ARPCache ResidentAgent this RoutingAlgorithm will
* use to determine the MAC address of the nexthop for outbound packets.
*/
public RoutingAlgorithm(ControlExecutive controlExec,
    ResidentAgent arpCache){

  flowRoutingTable = new FlowRoutingTable();// creates resident agent
  this.controlExec = controlExec;
  this.arpCache = arpCache;


controlExec.registerMessageProcessor((MessageProcessor)flowRoutingTable
);
    controlExec.registerMessageProcessor((MessageProcessor)arpCache);
    controlExec.registerCustomer(this);
    //add talker
    int channel_ID = ProtocolStackEvent.
    FROM_ROUTINGALGORITHM_TO_TRANSPORTINTERFACE_CHANNEL;
    try{
       controlExec.addTalkerToChannel(this,channel_ID);
       gui.sendText("Talking enabled on channel: " + channel_ID);
    }catch(ChannelException ce){
     gui.sendText(ce.toString());
    }
    //add listener
    channel_ID = ProtocolStackEvent.
      FROM_TRANSPORTINTERFACE_TO_ROUTINGALGORITHM_CHANNEL;
    try{
       controlExec.addListenerToChannel(this, channel_ID);
       gui.sendText("Listening to channel: "+channel_ID);
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }//try-catch

    Thread algorithmThread = new Thread(this,"Routing Algorithm");
    algorithmThread.start();
```

```
            gui.setTextField("I'm sleeping...");
        }//end RoutingAlgorithm()

    /**
   * The ControlExecutive calls this method to allow the RoutingAlgorithm
     * to set up the means of communication between itself and this new
     * Interface.
     * @param nextInterface The Interface this RoutingAlgorithm will add
     * to the list of Interfaces it monitors.
     */
    public void addInterface(Interface nextInterface){
      //add talker
      gui.sendText("Adding "+nextInterface.toString());
      int channel_ID = ProtocolStackEvent.
        FROM_ROUTINGALGORITHM_TO_INTERFACE_START_CHANNEL+
          interfaceCount;
      try{
         controlExec.addTalkerToChannel(this, channel_ID);
         gui.sendText("Talking enabled on channel: " + channel_ID);
      }catch(ChannelException ce){
       gui.sendText(ce.toString());
      }
      //add listener
      channel_ID = ProtocolStackEvent.
        ENQUEUING_INBOUND_PACKET_START_CHANNEL+
          interfaceCount;
      try{
         controlExec.addListenerToChannel(this, channel_ID);
         gui.sendText("Listening to channel: "+channel_ID);
      }catch(ChannelException ce){
        gui.sendText(ce.toString());
      }//try-catch
      interfaces.add(nextInterface);
      gui.sendText("Monitoring "+interfaces.size()+
        " interfaces");
      gui.sendText("Waiting...");
      interfaceCount++;
    }//addInterface()

    /**
     * Toe access an entry in the flowroutingtable
     * @param mid FlowRoutingtableEntry message formed by ONLY flow_id
     * @return  FlowRoutingTableEntry object
     */
    public synchronized  FlowRoutingTableEntry
      getFromFlowRoutingTable(Message mid){
      return (FlowRoutingTableEntry) this.flowRoutingTable.query(mid);
    }

    /**
     * To access the interfaces on the router
     * @return  Vector containing interfaces
     */
    public Vector getInterfaces(){
      return interfaces;
```

```
 }

 /**
  * Returns an array that contains the class names of the resident
  * agents that this Object desires to be a ResidentAgentCustomer of.
  * @return An array that contains the class names of the resident
  * agents that this Object desires to be a ResidentAgentCustomer of.
  */
 public String[] getAgentTypes(){
   return agentTypes;
 }//getAgentTypes()

/**
* When the ControlExecutive receives an agent that is to replace
* an existing agent, the ControlExecutive calls the replaceAgent
* method on each of the ResidentAgentCustomers of that ResidentAgent.
* @param agent The new ResidentAgent.
*/
 public void replaceAgent(ResidentAgent agent){
   if(agent.getClass().getName().equals(agentTypes[0])){
     gui.sendText("ARPCache is being replaced...");
     this.arpCache=agent;
     gui.sendText("New ARPCache installed...");
     gui.sendText(arpCache.toString());
   }else
   if(agent.getClass().getName().equals(agentTypes[1])){
     gui.sendText("FlowRoutingTable is being replaced...");
     //akkoc added line below
     this.flowRoutingTable = agent;
     gui.sendText("New FlowRoutingTable installed...");
     gui.sendText(flowRoutingTable.toString());
   }else{
     gui.sendText("RoutingAlgorithm is not a customer "+
       "of "+agent.toString());
   }
 }//replaceAgent()
 /**
  * The algorithm for determining which inbound queue to select from
  * next is contained here.  For this version of the SAAM router, the
  * algorithm merely visits the queues in round-robin fashion.  When
  * a packet is dequeued, it is sent to the routeInboundPacket method.
  */
 public void run(){

  //round-robin scheduler
  gui.sendText("Monitoring "+interfaceCount+" interfaces");

  while(true){
       gui.sendText("Looking in inbound queue's...");
       searchInboundQueues();
       gui.sendText(
         "***** All inbound queues empty, I'm going to sleep");
       try{
         synchronized(theLock){
           gui.sendText("Waiting...");
```

```
            while(!started)
              theLock.wait();
            gui.sendText("Resumed");
          }
        }catch(InterruptedException ie){
          gui.sendText(ie.toString());
        }
         }//while(started)
      }//run()


  /**
   * Searches all inbound queues for a packet.  Each queue is searched
   * until all packets in the queue have been removed.  All queues are
   * repeatedly searched until two conditions are met:
   * 1) All queues are empty and
   * 2) No Interface has enqueued a packet during the search.
   */
   private  void searchInboundQueues(){
     //Continue sweeping the queues and routing packets until all queues
     //are empty and no Interface sends a notification that a packet
     //is being enqueued.
     int queueID = Interface.INBOUND_QUEUE;
     gui.sendText("Searching... populatedQueues = "+populatedQueues);

     while ( populatedQueues > 0){

       for (int i = 0; i < interfaceCount; i++){
         Interface thisInterface = (Interface)interfaces.get(i);
         // round robin to achieve fairness
         if (!(thisInterface.isEmpty(queueID))){
           gui.sendText("** Number of packets in queue " + queueID+ "
                        is " + thisInterface.getPacketCount(queueID));
         gui.sendText("Dequeuing packet from "+
                      thisInterface.toString());
         byte [] packet = null;
         synchronized ( populatedQueueLock){
             packet = thisInterface.getPacket( Interface.INBOUND_QUEUE);
             if ( thisInterface.isEmpty(queueID) )
               populatedQueues--;
        }
           routeInboundPacket(packet);

        gui.sendText("** Number of packets remaining in queue " +
        queueID + " is " + thisInterface.getPacketCount(queueID));
        }//end if

        gui.sendText("SEARCH: Populated Queues = " + populatedQueues );

       }//end for

     }//while

    started = false;
}//searchInboundQueues()
```

```
    /**
     * Receives and processes SaamEvents received on the Channel that
     * has been designated as the communication Channel for traffic from
     * TransportInterface to the RoutingAlgorithm.
     * @param se The SaamEvent that contains the outbound packet.
     */
    public  void receiveEvent(SaamEvent se){ // akkoc made sync
     //this doesn't actually use the SaamEvent
      Interface thisInterface = (Interface)se.getTalker();
      synchronized ( populatedQueueLock){
       int numP = thisInterface.getPacketCount(Interface.INBOUND_QUEUE);
        if (numP == 1){
           populatedQueues++;
        }

      }// end of lock
      // Print out debugging information

    IPv6Packet packet = null;
    try{
      packet = new
       IPv6Packet(thisInterface.peekPacket(Interface.INBOUND_QUEUE));
      }catch(UnknownHostException uhe){};
      gui.sendText("\n Received Packet of Payload length = " +
      packet.getPayload().length);
      gui.sendText("Source: "+
      packet.getHeader().getSource().toString());
      gui.sendText("Dest: "+
      packet.getHeader().getDest().toString());

      gui.sendText("Number of populated queues = " + populatedQueues);

      if(!started){
       gui.sendText("packet arrived from : " + thisInterface+".
       Activating routing algorithm." + "\n");
         synchronized(theLock){
           started = true;
           theLock.notify();
         }
      }else{

      }//if(!started)

    }//packetReceived

/**
* This method answers the question - "Is this packet destined for
* an Interface on this router?".
* @param dataPacket The packet to be tested.
* @return True if the packet is destined for an Interface on this
router.
*/
public synchronized boolean isApplicationLayerPacket(IPv6Packet
```

```
                    dataPacket){

    for(int i=0;i<interfaces.size();i++){
      Interface thisInterface = (Interface)interfaces.get(i);
      if(dataPacket.getHeader().getDest().toString().equals(
        thisInterface.getID().getIPv6().toString()))
          return true;
    }
    return false;
  }//isApplicationLayerPacket()

  /**
  * The purpose of this method is to determine which Interface is
  * connected to the Interface this packet is travelling to next.
  * Compares the network portion of the nextHop provided with the
  * network portions of the IPv6Addresses of each of the the Interfaces
  * provided in the Vector.
  * @param interfaces The Vector of Interfaces on this router.
  * @param nextHop The IPv6Address to be compared.
  * @return The Interface that is connected to the Interface this packet
  * is travelling to next.
  */
  public Interface determineOutboundInterface(Vector interfaces,
    IPv6Address nextHop){
    byte[] nextHopBytes = nextHop.getAddress();
    for(int i=0;i<interfaces.size();i++){
      Interface thisInterface = (Interface)interfaces.get(i);
      //cycle through all interfaces
      //checking network address against nextHop.
      int match = 0;
      byte[] outboundInterfaceBytes =
        thisInterface.getID().getIPv6().getAddress();
      for(int index=0;index<bytesToCheck;index++){
        if((nextHopBytes[index]&0xFF)==
          (outboundInterfaceBytes[index]&0xFF)){
          match++;
        }//if
      }//for
      if(match==bytesToCheck){
        return thisInterface;
      }//if
    }//for
    return null;
  }//determineOutboundInterface()

/**
* Method is used to determine the correspongding interface on router
* for an IPV6Address by comparing the network address  portions
* @param adr IPv6Address that we are looking for interface with the
* matching network address.
* @return IPv6Address of interface with same network address on
router.
*/
public synchronized IPv6Address lookByNetworkAddress(IPv6Address adr){
```

```
    byte[] nextHopBytes = adr.getAddress();
    Enumeration eb = this.interfaces.elements();
    while( eb.hasMoreElements()){
      Interface thisInterface = (Interface)eb.nextElement();
      byte[] interfaceBytes =
             thisInterface.getID().getIPv6().getAddress();
       int match = 0;
       for(int index=0;index<bytesToCheck;index++){
         if((interfaceBytes[index]&0xFF)== (nextHopBytes[index]&0xFF)){
           match++;
         }//if
       }//inner for
       if(match== bytesToCheck){
         return thisInterface.getID().getIPv6();
       }//if
     }//while
     gui.sendText("no interface matches to searched nexthop address ");
     return null;

   }

 /**
 * Looks into the packet to determine whether it should be sent to
 * the application layer on this router or forwarded on to the next hop.
 * @param whichInterface The inbound Interface the packet was dequeued
 *        from.
 * @param packet The byte array that represents the packet.
 */
 public void routeInboundPacket(byte[] packet){
     packetCounter++;
     IPv6Packet dataPacket = null;
     try{
       dataPacket = new IPv6Packet(packet);
     }catch(UnknownHostException uhe){};
 gui.sendText("\nPacket#: "+packetCounter+", Size = " +
   packet.length + "; Payload length = " +
                         dataPacket.getPayload().length);
 gui.sendText("Source: "+
       dataPacket.getHeader().getSource().toString());
 gui.sendText("Dest: "+
       dataPacket.getHeader().getDest().toString());
 if(isApplicationLayerPacket(dataPacket)){
       gui.sendText("Application layer packet.");
       gui.sendText("Forwarding to Transport Interface.");
       ProtocolStackEvent event = new ProtocolStackEvent(
       toString(),this,ProtocolStackEvent.
       FROM_ROUTINGALGORITHM_TO_TRANSPORTINTERFACE_CHANNEL,
       dataPacket.getBytes());
       try{
           controlExec.talk(event);
       }catch(ChannelException tde){
           gui.sendText(tde.toString());
       }
 }else{
      forwardPacket(dataPacket);
```

```
  }
}//routeInboundPacket()

  /**
   * Checks to see whether or not an entry is in the ARPCache.
   * @param entry The ARPCacheEntry to be verified.
   * @return True if the entry is in the ARPCache.
   */
  public boolean checkARPCache(ARPCacheEntry entry){
      ARPCacheEntry resultEntry =
        (ARPCacheEntry)arpCache.query(entry);
      return resultEntry!=null;
  }//checkARPCache()

  //modified by [akkoc] according to new routingtables architecture
  /**
   * Forwards the outbound packet to the appropriate Interface
   * @param packet A byte array representation of the outbound packet.
   */
  public void forwardPacket(IPv6Packet packet){
    int flowID;
    IPv6Address dest;
    IPv6Header v6Header;

      v6Header = packet.getHeader();
      gui.sendText("\n Forwarding packet");
    flowID = v6Header.getFlowLabel();
    gui.sendText("Flow id: "+flowID);
    dest = v6Header.getDest();
    IPv6Address nextHop = null;
    byte sl = Interface.BEST_EFFORT_SL;

    //GX: First determine which routing table to look up
    if ( flowID != DATAGRAM_ROUTING ){
      Message message = (Message)(new FlowRoutingTableEntry(flowID));
      FlowRoutingTableEntry ent = (FlowRoutingTableEntry)
                                  flowRoutingTable.query(message);
      if(ent != null){
        nextHop = ent.getNextHop();
        sl = ent.getSL();
      }else{
        ServerTable st =  controlExec.getServerTable();
        if (st.hasEntryForDestination(flowID,dest)) {
          ServerTableEntry ste =
                 controlExec.getServerTable().getEntryByFlowId(flowID);
          RouterBoundCtrlChTable rbt = ste.getRouterBoundCtrlChTable();
          RouterBoundCtrlChTableEntry te = rbt.get(dest);
          nextHop = te.getNextHop();
          sl = Interface.CTRL_TRAFFIC_SL;
        }else {
          //use destination in IPv6 header
          nextHop = dest;
          sl = Interface.CTRL_TRAFFIC_SL;
        }
      }
```

```
    }
    else{
            // Datagram routing for Best effort or DiffServ traffic
    }

    //Now ARP to determine the MAC address of the next hop.
    //GX -- This part actually belongs to Interface.java

    Message message = (Message)(new ARPCacheEntry(nextHop));
    ARPCacheEntry entry = (ARPCacheEntry)arpCache.query(message);

    try{
      byte nextMAC = entry.getNextMAC();
      gui.sendText("nextMAC: "+nextMAC);
      outboundInterface = determineOutboundInterface(interfaces,
                                                     nextHop);
      gui.sendText("outboundInterface: "+outboundInterface);


if(v6Header.getSource().toString().equals(IPv6Address.DEFAULT_HOST)){
        gui.sendText("Setting Source");
        v6Header.setSource(outboundInterface.getID().getIPv6());
        packet.setHeader(v6Header);
     }

      gui.sendText("Source: " + v6Header.getSource());
      gui.sendText("Dest: " + v6Header.getDest());
      gui.sendText("Appending next hop MAC address " + (nextMAC&0xff));
      gui.sendText("Forwarding packet to: " + outboundInterface);

      byte[] outboundPacket = Array.concat(nextMAC,packet.getBytes());

      //send a SaamEvent to the appropriate outbound interface.
      //This SaamEvent contains the service level among other things.
      ProtocolStackEvent event = new
ProtocolStackEvent(toString(),this,

ProtocolStackEvent.getFromRoutingAlgorithmToInterfaceChannel(

interfaces.indexOf(outboundInterface)),outboundPacket,sl,nextHop);
      try{
        controlExec.talk(event);
      }
      catch(ChannelException tde){
        gui.sendText(tde.toString());
      }
    }
    catch(NullPointerException npe){
      gui.sendText("Next Hop is not in the ARPCache");
      gui.sendText("Packet Dropped\n");
    }
      }//forwardPacket()

/**
```

```
* Returns a <code>String</code> representation of this object
* @return The <code>String</code> representation of this object
*/
public String toString(){
  return "Routing Algorithm";
}//toString()
}
```

```
//-----------------------------------------
// Filename : Server.java
// Feb 2000[akkoc]    - modified
// 01Aug99 [Vrable]   - Created
// Project   : SAAM
//-----------------------------------------------------------------------


package saam.server;

import saam.EmulationTable;
import saam.Translator;
import saam.*;

import saam.net.*;
import saam.message.*;
import saam.control.*;
import saam.event.*;
import saam.router.*;
import saam.util.*;
import java.net.*;
import java.util.*;
import java.io.*;

/**
* The <em>Server</em> is an object within the SAAM architecture that
* maintains a picture of the network for use in assigning flows.
*/

public class Server implements  Runnable{

/** Contains what is known about the network. */
private PathInformationBase PIB;

/** Enables the Server to receive and send particular types of
messages. */
private ControlExecutive controlExec;

/** A maximum number of hops that a search for different paths may
take. */
private int Hmax = 4;

/**
* Used to lookup what flow id should be used to send out control
messages
* to specified routers.
*/
private Hashtable flowLookUp = new Hashtable();

/** Used to assign the right number of service level pipes to
interfaces in
* this SAAM region. Only used during initialization
*/
private int numOfServiceLevels = 4;
```

```
/**
* The value assigned to flow ids that can not be supported. This should
* switched over to 0 as soon as routers are converted.
*/
  public static int FLOWNOTSUPPORTABLE = 99;

  public static int INITIALDELAY = 0;
  public static int INITIALLOSSRATE = 0;
  public static int INITIALTHROUGHPUT = 10000;

  public static int RETURNFLOWDELAY = 50;
  public static int RETURNFLOWLOSSRATE = 50;
  public static int RETURNFLOWTHROUGHPUT = 1000;

  public static int ROUTERNOTINPIB = 0;

  public static int NOSUPPORTABLEPATHINPIB = 0;

  public static int SERVERNODEID = 1;

  public static int FLOWTOSERVER = 0;

  public static int PSUEDORANDOMSOURCEPORT = 8000;

  public static int INITIALPATHID = 0;

  public static int INITIALHEIGHTOFSEARCH = 1;
  public static int INCREMENTATIONOFSEARCH = 1;
  public static int DESTINATIONNODE = 0;

  public static int INITIALZERO = 0;


  /** Defines with the appropriate IPv6 address of this server. */
  private String serverIPv6 = "99.99.99.0.0.0.0.0.0.0.0.0.0.0.0.1";

  /** Time when the all possible paths were found. */
  private long timeOfLastPIBBuild = System.currentTimeMillis();

 /**
 * The amount of time that we want to have between rebuilding of paths.
 */
  private long timeBetweenPIBBuilds = 120000; // 2 minutes (or 120 sec)

  /** A boolean that will allow the showing of comments. */
  private boolean showComments = true;

  private SAAMRouterGui gui;

  // 2000 akkoc added
  private int sequenceNumber = 1;
  private static final int CTS = 0;     //SINCE ITS SERVER BY ITSELF
  private int hopCount;
```

```
  private static byte serverType;
  private static int flowId;
  private static byte metricType;
  private static int cycleTime;
  private static int globalTime;

  private IPv6Address ServerId;


  /**
  * Constructs a server that use a specified type of Path Information
  * Base.  The PIB may be in the form of a database structure (which
  * requires an existing ODBC configured local database) or a class
  * object structure. The cont executive is the interface to the IPv6
  * protocol stack, in order for messages to flow to and from network.
  * The final step taken is the deletion of all existing data, which is
  *important only in a database structure since a class object
  *structure is
  * volatile.
  * @param type The type of structure that the PIB is to assume.
  * @param controlExec The control executive that will exchange message
  * with this server.
  */
  public Server(String type, ControlExecutive controlExec){
    if (type == "database")
      PIB = new DatabaseStructure();
    else
      PIB = new ClassObjectStructure();

    this.controlExec = controlExec;
    gui=new SAAMRouterGui("Server");

    ServerId = controlExec.getRouterId();

    PIB.deleteAllData();
  }

//*********************************************************************
********
// These methods handle external network communications from routers
//*********************************************************************
*******/

  public void processHello(Hello hello) {

    long start, finish;
    Vector interfaces;
    int node_id = INITIALZERO;
    InterfaceID myInterface;
    int bandwidth = INITIALZERO;
    IPv6Address address = new IPv6Address();
    Vector IPv6Addresses = new Vector();
    boolean newRouter = true;
    FlowRequest myFlowRequest = new FlowRequest();
```

```
    // capture the start time of processing a hello
    start = System.currentTimeMillis();

    // produce a vector of IPv6Addresses
    interfaces = hello.getInterfaceIDs();
    for (int i = INITIALZERO; i < interfaces.size(); i++){
      address = ((InterfaceID)interfaces.elementAt(i)).getIPv6();
      IPv6Addresses.addElement(address);
    }

//check if router exists and if so, return it's node id, else return 0
  node_id = PIB.doesRouterExist(IPv6Addresses);

  // if the router does not exist in PIB
  if (node_id == ROUTERNOTINPIB){
     // assign it a new node id
     node_id = PIB.getNewNodeId();
   } else {
     newRouter = false;
   }

   // run through all of the LSA interfaces
   for (int i = INITIALZERO; i < interfaces.size(); i++) {
     myInterface = (InterfaceID)interfaces.elementAt(i);
     address = myInterface.getIPv6();
     // if a new interface is not found in the PIB, then ...
     if (!PIB.doesInterfaceExist(address)){
       bandwidth = myInterface.getBandwidth();
       address = myInterface.getIPv6();
       // if the link is not contained in the PIB, then add it
       if (!PIB.doesLinkExist(address)){
         PIB.addLink(address, bandwidth);
       }
       // now add the interface between the node and the link
       PIB.addInterface(node_id, address);
       // now add each service level pipe
       for (int service_level = 0; service_level < numOfServiceLevels;
                                                   service_level++){
         PIB.addSLP(address, service_level, INITIALDELAY,
         INITIALLOSSRATE, INITIALTHROUGHPUT);
       }
     } // end if
   } //end interfaces for

   // capture the hello processing finish time
   finish = System.currentTimeMillis();
     gui.sendText("Server: processHello: Time required = "
       +(finish-start)+" milliseconds.");


     // rebuild all possible paths
     findAllPossiblePaths();

     // determine effective QoS of each path
```

```
      determineEffectiveQoSForPaths();

      // construct a new flow to this router
      try{
        myFlowRequest = new
            FlowRequest(IPv6Address.getByName(serverIPv6),
            address,System.currentTimeMillis(),
            RETURNFLOWDELAY,
            RETURNFLOWLOSSRATE,RETURNFLOWTHROUGHPUT);
      } catch(UnknownHostException uhe){
        System.err.println("Server: main:UnknownHostException:" + uhe);
      }
      processFlowRequest(myFlowRequest);

  } //end processFlowRequest


  /**
* Receives link state advertisement messages from router and processes
* service level pipe status information that they contain. It begins by
* checking to see if a router with the interface address described by
this
* LSA is known to the PIB. If such a router is known to exist, it then
* checks to see if the service level pipe described by this LSA is
known to
* the PIB. If the service level pipe is known, then update its status.
* Otherwise, add the SLP with the specified QoS characteristics.
Finally,
* update the effective QoS for the paths that pass over this service
* pipe by calling the determineEffectiveQoSForPaths().
* @param router A representation of a router as defined by an LSA.
*/
public void processLSA(LinkStateAdvertisement LSA) {

    long start, finish;
    int node_id = INITIALZERO;
    int bandwidth = INITIALZERO;
    byte service_level = 0;
    int delay = INITIALZERO;
    int loss_rate = INITIALZERO;
    int utilization = INITIALZERO;
    Vector interfaces = new Vector(3,1);
    Vector SLPs = new Vector(3,1);
    IPv6Address link_id;
    IPv6Address address;
    Vector IPv6Addresses = new Vector(1,1);

    // capture the start time of processing an LSA
    start = System.currentTimeMillis();

    // produce a one element vector of IPv6Addresses
    address = LSA.getMyIPv6();
    IPv6Addresses.addElement(address);

// check if router exists and if so, return it's node id, else return 0
```

```
    node_id = PIB.doesRouterExist(IPv6Addresses);

    // if the router does exist in PIB, then so does the interface...
    if (node_id != ROUTERNOTINPIB){

      service_level = LSA.getServiceLevel();
      delay = LSA.getDelay();
      loss_rate = LSA.getLossRate();
      utilization = LSA.getUtilization();

      if (showComments){
        gui.sendText("Server: processLSA: node_id = " + node_id
          + ", address = " + address + ", SL = "+service_level
          +", D = " +delay+ ", LR = "+loss_rate
          +", U = "+utilization);
      }

      // if this SLP is defined, then just update its status
      if(PIB.doesSLPExist(address, service_level))
      {
        PIB.updateSLP(address, service_level, delay,
                              loss_rate, utilization);
      }
      // otherwise, insert it
      else {
        PIB.addSLP(address, service_level, delay,
                              loss_rate, utilization);
      } // end else
    } // end if
    else { //do nothing
    }

    // capture the LSA processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: processLSA: Time required = "
      +(finish-start)+" milliseconds.");

    // revise the effective QoS of paths made up of this SLP
    determineEffectiveQoSForPaths(address,service_level);

  } //end processLSA


/**
* Receives and processes flow requests from applications. It begins
* by finding a source and a destination router. Routers may be where
* the applications are residing themselves, which is our standard
situation.
* The application could, however, reside on some host that is not
registered
* with the PIB as a router. In this case, the appropriate source or
* destination router would be a router connected to the same link. <p>
* The PIB is checked to ensure that there is the effective QoS
available on
* some path to satisfy the request. If satisfactory path is found, new
```

```
* unique flow id is assigned and this new flow is associated with that
path.
* Each router in the path is retrieved and a new flow routing table
entry is
* sent to each. If no path can provide the requested level of QoS, then
* flow is assigned to zero, which will be interpreted by IPv6 as best
effort
* traffic. Finally, a flow response is sent back to the application to
* inform it of its assigned flow id. If the flow id that is return is
zero,
* it will be the application's responsibility to either lower it QoS
request
* or to send its traffic as best effort.
* @param flow_request The message requesting the establishment of a
flow.
*/
public void processFlowRequest(FlowRequest flow_request) {

    /** A vector of slp_sequence information for a path. */
    Vector slps_in_path;
    SLPSequence currentSLPSequence,nextSLPSequence = new SLPSequence();
    int SLP_source_router, SLP_destination_router, service_level;
    IPv6Address link_id = new IPv6Address();
    IPv6Address next_hop;
    IPv6Address sourceAddress;
    int source_router, destination_router, path_id,
      flow_id=FLOWNOTSUPPORTABLE;
    long start, finish;

    // capture the start time of processing a flow request
    start = System.currentTimeMillis();

    // find a router on the same subnet as the source host
    source_router =
      PIB.findARouterOnLink(flow_request.getSourceInterface());

    // find a router on the same subnet as the destination host
    destination_router =
      (PIB.findARouterOnLink(flow_request.getDestinationInterface()));

    path_id = PIB.getPathThatCanSupportFlowRequest(source_router,
                                                   destination_router,
                                                   flow_request);

    // if a path can support this request, then...
    if(path_id != NOSUPPORTABLEPATHINPIB){

      // assign a flow id to the request
      flow_id = PIB.getNewFlowId(path_id,source_router,
                                 destination_router,
                                 flow_request);

      // determine each router in path
      // transmit Flow Routing Table Entry to it
      slps_in_path = PIB.getSLPSequenceOfPath(path_id);
```

```
    // for each router in the path, send a FRTE update
    for (int index = INITIALZERO;
                    index < slps_in_path.size(); index++){
     // assign new slp sequence object
     currentSLPSequence = (SLPSequence)slps_in_path.elementAt(index);

     // if not the last link..
      if (index+1 != slps_in_path.size()){
        nextSLPSequence =
        (SLPSequence)slps_in_path.elementAt(index+1);
      }

      // retrieve values from this object
      SLP_source_router = currentSLPSequence.getSourceRouter();
      link_id = currentSLPSequence.getLinkId();
      service_level = currentSLPSequence.getServiceLevel();

      // if not the last link...
      if (index+1 != slps_in_path.size()){
        SLP_destination_router = nextSLPSequence.getSourceRouter();
      } else {
        // else it is the destination node of the flow
        SLP_destination_router = destination_router;
      }
      // determine destination address for next hop
      next_hop = PIB.getInterfaceAddress(
                    SLP_destination_router, link_id);

      // determine source address
      sourceAddress = PIB.getInterfaceAddress(
                     SLP_source_router, link_id);

      // send the flow routing table entry update
      sendFRTEUpdate(sourceAddress, flow_id,
                    next_hop, service_level);

    } // end for

  } // end if

  //give routers time to finish updating tables
  try{
    Thread.sleep(2000);
  }catch(InterruptedException ie){
    gui.sendText(ie.toString());
  }
   // if the source of this flow is the server,
  if (source_router == SERVERNODEID) {
    // then add this new flow to hash table for later lookup
    if (showComments){
      gui.sendText("Server: processFlowRequest: use flow "+flow_id
        +" to send to node "+destination_router);
    }
    if (destination_router == SERVERNODEID) {
      flow_id = FLOWTOSERVER;
```

```
      }
   flowLookUp.put(new Integer(destination_router),new Integer(flow_id));
   }

   sendFlowResponse(flow_request, flow_id);

     // capture the flow request processing finish time
     finish = System.currentTimeMillis();
     gui.sendText("Server: processFlowRequest: Time required = "
       +(finish-start)+" milliseconds.");
   }

   /**
    * Receives flow termination from routers and then processes them.
    */
   public void receiveFlowTermination() { }


//*********************************************************************
// These methods handle external network communications to routers
//*********************************************************************

/**
*Sends flow routing table entry update message to router. This message
* provides router the required information to forward packets based on
* its flow id.
* @param sourceAddress The router that will receive the FRTE update.
* @param flow_id The id assigned to the flow in question.
* @param next_hop The IPv6 address of the next node in the path.
* @param service_level The service level that this flow is assigned to.
*/
public void sendFRTEUpdate(IPv6Address sourceAddress, int flow_id,
                                    IPv6Address next_hop,
                                    int service_level) {
    if(showComments){
      gui.sendText("Server: sendFRTEUpdate: flowLookUp hashtable:");
      gui.sendText(""+flowLookUp);
    }
    FlowRoutingTableEntry myFRTE = new FlowRoutingTableEntry(flow_id,
                                          (byte)service_level,next_hop);
    int sourcePort = PSUEDORANDOMSOURCEPORT;

//controlExec.listenToRandomPort(this);
    short destPort = ControlExecutive.SAAM_CONTROL_PORT;
    IPv6Address destHost = sourceAddress;
    // take steps to determine what flow id to send the packet on
    Vector interfaces = new Vector();
    interfaces.addElement(destHost);
    int destNodeId = PIB.doesRouterExist(interfaces);
    int flowIdToSendItOn = ((Integer)flowLookUp.get
                                          (new
Integer(destNodeId))).intValue();
    try{
      controlExec.send(this,myFRTE, flowIdToSendItOn,
```

```
(short)sourcePort,
destHost, destPort);
    } catch (FlowException fe){
      System.err.println(fe.toString());
    }
    if (showComments){
      gui.sendText("Server: sendFRTEUpdate: FRTE for flow " + flow_id
        + " sent to interface "+sourceAddress);
      gui.sendText("          with next hop= "+next_hop
        +" on service level "+service_level+" via flow
"+flowIdToSendItOn);
    }
  }

/**
 * Sends a flow response to the requesting application to notify it of
 * its newly assigned flow id. Flow id zero is used to indicate that
 * flow cannot be supported. Once a flow response message is
instantiated and
 * a source and destination port is defined, control executive's send()
 * is called to send it to the destination host.
 * @param flow_request The flow request message that was received.
 * @param flow_id The flow id that is assigned to the flow request.
 */
 public void sendFlowResponse(FlowRequest flow_request, int flow_id){
    if(showComments){
      gui.sendText("Server: sendFlowResponse: flowLookUp hashtable:");
      gui.sendText(""+flowLookUp);
    }
    FlowResponse response = new
         FlowResponse(flow_request.getTimeStamp(),
         flow_id);
    int sourcePort = PSUEDORANDOMSOURCEPORT;

    short destPort = ControlExecutive.SAAM_CONTROL_PORT;
    IPv6Address destHost = flow_request.getSourceInterface();
    // take steps to determine what flow id to send the packet on
    Vector interfaces = new Vector();
    interfaces.addElement(destHost);
    int destNodeId = PIB.doesRouterExist(interfaces);
    int flowIdToSendItOn = ((Integer)flowLookUp.get
                                          (new
Integer(destNodeId))).intValue();
    try{
      controlExec.send(this, response,
      flowIdToSendItOn, (short)sourcePort,
     destHost, destPort);
    }catch(FlowException fe){
      System.err.println(fe.toString());
    }
    if (showComments){
      gui.sendText("Server: sendFlowResponse: Flow response "
        + response + " from SourcePort: "+sourcePort+" to "+destHost
        + " sent via flow "+flowIdToSendItOn);
    }
```

```
}

//*********************************************************************
// These methods handle internal manipulation of data describing
network status
//*********************************************************************

/**
* Determines all of the possible paths that exist between any source
* destination router in the network. This determination is based on
* physical definition of the network that is provided by the hello
* received from the routers and stored within the PIB. The paths that
* found are then recorded in the PIB for fast assignment of flows
later.<p>
* All node ids are first retrieved from PIB. For each service level, we
* build an array of parents of each node. A parent is node that is
directly
* connected. Those directly connected nodes would have service level
pipes
* that would need to be passed through to get to the child node in
question.
* This parent array is used to populate a path table. Each node id is
* assigned as the final destination of path and all of different paths
* are then found by working out from this destination. For each of
these
* destination nodes, a call is made to processPath() to find all the
valid
* paths that go to this destination node. We make the call with a
specified
* height of search of 1.
*/
  public void findAllPossiblePaths() {
    long start, finish;
    int NumberOfRouters;
    int max_slp_id = INITIALZERO;
    /** A count of the highest path id assigned so far. */
    int max_path_id = INITIALZERO;
    int service_level = INITIALZERO;

    /** A vector of the routers that are known by the db. */
    Vector V = new Vector();

    /** A vector of parent routers for each given destination router.
    Hashtable parent;

    // capture the start time of processing a path data
    start = System.currentTimeMillis();

    // reset the maximum path id assigned so far to zero
    max_path_id = INITIALPATHID;

    V = PIB.getAllRouterIds();

    //retrieve COUNT of routers
    NumberOfRouters = V.size();
```

```
    //find all possible paths for each service level
    max_slp_id = (new Integer(PIB.findMaxServiceLevel())).intValue();
    for (service_level = INITIALZERO; service_level <= max_slp_id;
    service_level++){

    //build parent array of each SLP at this service level
      parent = PIB.getParents(V, service_level);

      //populate path table
      for (int index = INITIALZERO; index < NumberOfRouters; index++){
        int heightOfSearch = INITIALHEIGHTOFSEARCH;
        int aPath[] = new int[Hmax + INCREMENTATIONOFSEARCH];
        aPath[DESTINATIONNODE] =
                          ((Integer)V.elementAt(index)).intValue();
        processPath(parent, aPath, heightOfSearch,service_level);
      }
    }

    // capture the path data processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: findAllPossiblePaths: Time required = "
      +(finish-start)+" milliseconds.");

    timeOfLastPIBBuild = finish;

  }

/**
* Processes all valid paths that arrive at destination node within some
* range of hops. For each parent of the node at the distance of
* heightOfSearch from destination, check is made to ensure that adding
* this new parent will cause no cycle. If this checks out, then that
parent
* can be added and a new path can be assigned. The service level pipes
* this new path are identified and their sequence numbers in this path
* recorded to the PIB. Next, a check is made to see if the height of
* search is less than the server's max search height of Hmax. If it is
* the method recursively calls itself with an incremented
heightOfSearch
* variable.
* @param parent Contains each router and a list of other
* routers that are directly attached to them.
* @param aPath[] An array contain a path from a source node,
* aPath[heightOfSearch], to a destination node, aPath[0].
* @param heightOfSearch The number of nodes in the path so far.
* @param service_level The level of service assigned to a flow.
*/

public void processPath(Hashtable parent,
                          int aPath[], int heightOfSearch,
                          int service_level){
    IPv6Address link_id;
    int justARouter;
    int sequence_number;
```

```
    int path_id;
    Enumeration W = ((Vector)parent.get(
                    new Integer(aPath[heightOfSearch-1]))).elements();
    while (W.hasMoreElements()) {
      justARouter = ((Integer)W.nextElement()).intValue();
      if (causeNoCycle(aPath, heightOfSearch, justARouter)) {

        // assign this router as the source in this path
        aPath[heightOfSearch] = justARouter;

        // record the new path id, etc.
        path_id = PIB.getNewPathId(justARouter,
                      aPath[DESTINATIONNODE]);

        // run through the SLP's and record their sequence
     for (int index = heightOfSearch; index >DESTINATIONNODE; index--){

       // determine link_id of this SLP
       link_id = PIB.getLinkBetween(aPath[index],
                            aPath[index-INCREMENTATIONOFSEARCH]);

          // assign the SLP its sequence number
          sequence_number = heightOfSearch - index;
          PIB.assignSLPSequence(service_level, aPath[index],
            link_id, path_id, sequence_number);
        }
      if (heightOfSearch < Hmax) { processPath(parent, aPath,
                                heightOfSearch+INCREMENTATIONOFSEARCH,
                                service_level);
        }
      }
    }
    if (showComments){
      gui.sendText("Server: processPath: paths at depth of
      "+heightOfSearch
      +" from node "+aPath[DESTINATIONNODE]+" is completed.");
    }
  }

/**
* Checks to ensure that addition of a specified new node to a specified
* path does not result in a cycle being created. This check is
completed by
* the new node is already a member of the list of nodes in the path
already.
* @param aPath[] An array contain a path from a source node,
* aPath[heightOfSearch], to a destination node, aPath[0].
* @param heightOfSearch The number of nodes in the path so far.
* @param justARouter The proposed next node in for a new path.
* @returns noCycles True if no cycles are created by the addition of
* justARouter.
*/
public boolean causeNoCycle(int aPath[], int heightOfSearch,
                            int justARouter){
    boolean noCycles = true;
```

```
    for (int index = INITIALZERO; index < heightOfSearch; index++){
      if (justARouter == aPath[index]){
        if (showComments){
          gui.sendText("Server: causeNoCycle: adding "+justARouter
            +" to get to "+aPath[DESTINATIONNODE]+" via "
            +aPath[heightOfSearch-INCREMENTATIONOFSEARCH]
            +" at a height of "+heightOfSearch+" caused cycle!");
        }
        return noCycles = false;
      }
    }
    if (showComments){
      gui.sendText("Server: causeNoCycle: adding "+justARouter
        +" as hop #"+heightOfSearch+" to get to
        "+aPath[DESTINATIONNODE]
        +" via "+aPath[heightOfSearch-INCREMENTATIONOFSEARCH]
        +" does not cause cycle.");
    }
    return noCycles;
  }

/**
* Determines what effective QoS on each path in the PIB is. For each
* path, service level pipes that compose it are retrieved. Then, for
* each of these service level pipes, we total up delay and loss rate.
* The effective throughput remaining is determined by finding minimum
* difference between observed throughput and the target throughput of
* each service level pipe.
*/
public void determineEffectiveQoSForPaths(){
    long start, finish;
    Vector path_ids;
    Integer myPathId;
    Vector SLPs;
    SLP mySLP;
    int totalDelay = INITIALZERO, totalLossRate = INITIALZERO,
      throughput = INITIALZERO, targetThroughput = INITIALZERO,
      throughputRemaining = INITIALZERO,
      minThroughputRemaining = INITIALZERO;

    // capture the start time of processing a path data
    start = System.currentTimeMillis();

    // for each path
    path_ids = PIB.getAllPathIds();

    for (int index1 = INITIALZERO; index1 < path_ids.size(); index1++){

      // for each path
      myPathId = (Integer)path_ids.elementAt(index1);

      SLPs = PIB.getSLPsOfPath(myPathId.intValue());

      for (int index2 = INITIALZERO; index2 < SLPs.size(); index2++){
```

```
        mySLP = (SLP)SLPs.elementAt(index2);

        // add delay to total delay
        totalDelay = totalDelay + mySLP.getDelay();

        // add loss rate to total loss rate
        totalLossRate = totalLossRate + mySLP.getLossRate();

        // find min throughput
        throughput = mySLP.getThroughput();

        targetThroughput = mySLP.getTargetThroughput();

        throughputRemaining = targetThroughput - throughput;
        if (throughputRemaining < minThroughputRemaining ||
                                  minThroughputRemaining == INITIALZERO){
          minThroughputRemaining = throughputRemaining;
        }

      }


PIB.setEffectiveQoSOfPath(myPathId.intValue(),totalDelay,totalLossRate,

minThroughputRemaining);
      totalDelay = INITIALZERO;
      totalLossRate = INITIALZERO;
      minThroughputRemaining = INITIALZERO;
    }

    // capture the path data processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: determineEffectiveQoSForPaths: Time required
= "
      +(finish-start)+" milliseconds.");

  }

/**
* Determines the effective QoS for just those paths that pass over the
* specified service level pipe. For each path, service level pipes that
* compose it are retrieved. Then, for each of these service level
pipes, we
* total up delay and loss rate. The effective throughput remaining is
* determined by finding the minimum difference between the observed
* throughput and the target throughput of each service level pipe.
* @param address Address of interface containing this service level.
* @param service_level The service level of this SLP.
*/
public void determineEffectiveQoSForPaths(IPv6Address address, int
service_level){
    long start, finish;
    Vector path_ids;
    Integer myPathId;
```

```
Vector SLPs;
SLP mySLP;
int totalDelay = INITIALZERO, totalLossRate = INITIALZERO,
  throughput = INITIALZERO, targetThroughput = INITIALZERO,
  throughputRemaining = INITIALZERO, minThroughputRemaining =
  INITIALZERO;

// capture the start time of processing a path data
start = System.currentTimeMillis();

// for each path
path_ids = PIB.getAllPathIdsThatTraverseSLP(address,
                                            service_level);

for (int index1 = INITIALZERO; index1 < path_ids.size(); index1++){

  // for each link
  myPathId = (Integer)path_ids.elementAt(index1);

  SLPs = PIB.getSLPsOfPath(myPathId.intValue());

  for (int index2 = INITIALZERO; index2 < SLPs.size(); index2++){

    mySLP = (SLP)SLPs.elementAt(index2);

    // add delay to total delay
    totalDelay = totalDelay + mySLP.getDelay();

    // add loss rate to total loss rate
    totalLossRate = totalLossRate + mySLP.getLossRate();

    // find min throughput
    throughput = mySLP.getThroughput();

    targetThroughput = mySLP.getTargetThroughput();

    throughputRemaining = targetThroughput - throughput;
    if (throughputRemaining < minThroughputRemaining ||
       minThroughputRemaining == INITIALZERO){
       minThroughputRemaining = throughputRemaining;
    }

  }


  PIB.setEffectiveQoSOfPath(myPathId.intValue(),
        totalDelay,totalLossRate,
        minThroughputRemaining);
  totalDelay = INITIALZERO;
  totalLossRate = INITIALZERO;
  minThroughputRemaining = INITIALZERO;
}

// capture the path data processing finish time
finish = System.currentTimeMillis();
```

```
  gui.sendText("Server: determineEffectiveQoSForPaths: Time required ="
      +(finish-start)+" milliseconds.");
  }
  /**
   * Returns the String representation of this Server.
   * @return The String representation of this Server.
   */
  public String toString(){
    return "Server";
  }  //

  //methods below are added by akkoc
  /**
   * Creates thread for dcm sending from the server.
   * @return void.
   */
  public void autoConfig() {
    Thread configThread = new Thread(this,"AutoConfig");
    configThread.start();
  }//end of autoconfig

/**
* Triggers DCM sending and refreshment channels of SAAM region
* @return void.
*/
public void run(){
  gui.sendText("\n Server  will send first DCM after 30 secs");
  try{
     gui.sendText("thread is sleeping now ");
     Thread.sleep(30000);
     gui.sendText("thread woke up after 30  secs so start sending ");
  }catch(InterruptedException ie){}


 while(true) {

   try{

     Vector tableEntries =
            controlExec.getEmulationTable().getEmTable();
     Enumeration es = tableEntries.elements();
     while( es.hasMoreElements()){
       EmulationTableEntry ent = (EmulationTableEntry)
                                        es.nextElement();
       //destination adress determined from emulationtable entry
       IPv6Address des = new
       IPv6Address(ent.getNextHopIPv6().getAddress());
       gui.sendText(" Destination of DCM is "+des.toString());
       byte[] nextHopBytes = des.getAddress();
       Vector interfaces = new Vector();
       interfaces = this.controlExec.getInterfaces();

       IPv6Address sInt;
       for(int i=0;i<interfaces.size();i++){
         Interface thisInterface = (Interface)interfaces.get(i);
```

```
//cycle through all interfaces checking network address against
//nextHop.
int match = 0;
byte[] outboundInterfaceBytes =
thisInterface.getID().getIPv6().getAddress();
int bytesToCheck = 5;

      for(int index=0;index<bytesToCheck;index++){
          if((nextHopBytes[index]&0xFF)==
          (outboundInterfaceBytes[index]&0xFF)){
              match++;
           }//if
         }//inner for

         if(match== bytesToCheck){
           sInt = new
           IPv6Address(thisInterface.getID().getIPv6().getAddress());
           sendDown(sInt,des);
         }//if
       }//outer for

     }// end while
   }catch(UnknownHostException e){
     gui.sendText(e.getMessage)+"inside catch of DCM start Emtable  ");
   }//try-catch

   try{
     Thread.sleep(this.cycleTime); //from demostation
   }catch(InterruptedException ie){
     gui.sendText("thread sleep problem");
   }

  }//end of while providing continues DCM sending

 }// end run()


  /**
   * Retruns flowid of server.
   * @return ind serverflow id.
   */
  public int getServerFlowId(){
    return flowId;
  }

  /**
   * Returns type of server(0-> for Primary, 1-> for Backup )
   * @return byte value.
   */
  public byte  getServerType(){
     return serverType;
  }

  /**
```

```
   * Method to send the DCM message using controlExecutive sendDCM
method
   * @return void.
   */
  public void sendDown(IPv6Address srcInt,IPv6Address des) {

    DCM myDCM = new
DCM(flowId,ServerId,metricType,srcInt,CTS,globalTime,

getSeguenceNumberForDcmSending());
  gui.sendText("DCM with SQ is sent
  "+this.getSeguenceNumberForDcmSending());
  setSeguenceNumberForDcmSending();
  short sourcePort = ControlExecutive.SAAM_CONTROL_PORT;
  short destPort = ControlExecutive.SAAM_CONTROL_PORT;

  try{
       controlExec.sendDCM(this, myDCM, getServerFlowId(),
       sourcePort,des, destPort);
       gui.sendText("DCM has been sent");
    }catch(Exception fe){
       System.err.println(fe.toString());
    }

  }//end sendDown()

  /**
   * Method for setting proper value to put in DCM message for sequence
   * number field
   * @return void.
   */
  private void setSeguenceNumberForDcmSending(){
    sequenceNumber++;
    if(sequenceNumber == 65535) sequenceNumber = 0;
  }

   /**
   * Method for returning current sequence number value
   * @return int value.
   */

  private int getSeguenceNumberForDcmSending(){
    return  sequenceNumber;
  }


 /**
 * To receive required values from demosation for server settings
 * Also this method is used for server to place an entry for itself
 * in the servertable
 * @return void.
 */
  public synchronized void processConfiguration (Configuration con){
    serverType = con.getServerType();
    flowId = con.getFlowId();
```

```
    metricType = con.getmetricType();
    cycleTime = con.getCycleTime();
    globalTime = con.getGlobalTime();

    AutoConfigurationExecutive ace =
                    controlExec.getAutoConfigurationExecutive();
    ace.createNewServerInformation(flowId,controlExec.getRouterId());
 } // end processConfigurtaion

}//end of class
```

LIST OF REFERENCES

1. W.Stallings, "High-Speed Networks TCP/IP and ATM Design Principles", Prentice Hall, 1998
2. Xie, G., Hensgen, D., Kidd, T., and Yarger, J., "Efficient Management of Integrated Services Using A path Information Base," [http://ww.cs.nps.navy.mil/people/faculty/xie/pub]. 14 May 1998.
3. L. L. Peterson, B.S. Davie, "Computer Networks, A System Approach",Morgan Kaufmann, 1996
4. J. F. Kurose, K. W. Ross, "Computer Networking, A Top-Down Approach Featuring the Internet", Addison Wesley, 1999
5. S. Keshav, " An Engineering Approach to Computer Networking", Addison Wesley, 1997
6. T. Lammle, D. Porter, J. Chellis " CCNA Cisco Certified Network Associate Study Guide", Network Press Sybex, 1998
7. Vrable, Dean, Yarger, John, "The SAAM Architecture: Enabling Integrated Services", Master Thesis September 1999, Computer Science Department Naval Postgraduate School
8. Xie, G., Hensgen, D., Kidd, T., and Yarger, J., "SAAM: An Integrated Network Architecture for Integrated Services," paper presented at the 6th IEEE/IFIP International Workshop on Quality of Service, Napa, CA, [http://ww.cs.nps.navy.mil/people/faculty/xie/pub].. May 1998

9. Kati, E., "Fault-Tolerant Approach For Deploying Server Agent-Based Active Network Management (SAAM) Server In Windows NT Environment To Provide Uninterrupted Services To Routers In Case Of Server Failure(s)," Master Thesis March 2000, Computer Science Department Naval Postgraduate School
10. Cormen, T.H., Leiserson C.E., Rivest R.L., "Introduction to Algorithms", McGraw Hill, 1998

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library.....2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Engineering and Technology Curriculum (Code 34).....2
   Naval Postgraduate School
   Monterey, CA 93943-5101

4. Genelkurmay Baskanligi.....1
   Personel Baskanligi
   Bakanliklar
   Ankara, TURKEY

5. Kara Kuvvetleri Komutanligi.....1
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY

6. Kara Kuvvetleri Komutanligi.....1
   Kutuphanesi
   Bakanliklar
   Ankara, TURKEY

7. Kara Harp Okulu.....2
   Kutuphanesi
   Dikmen
   Ankara, TURKEY

8. Chairman, Code CS.....1
   Naval Postgraduate School
   Monterey, CA 93943-5101

9. Prof.. Geoffrey Xie, Code CS/Xg.....1
   Naval Postgraduate School
   Monterey, CA 93943-5100

10. CDR. Deborah Kern, Code CS/kh.....1
Naval Postgraduate School
Monterey, CA 93943-5100

11. 1st LT Huseyin UYSAL.....1
Genel Kurmay Baskanligi
Bilgi Sis. D. Bsk.
Ankara, TURKEY

12. 1st LT Hasan AKKOC.....1
Kara Harp Okulu
Ogretim Baskanligi
Ankara, TURKEY